SecML

Release 0.13.post1

PRALab

Aug 06, 2020

USER GUIDE

SecML is an open-source Python library for the **security evaluation** of Machine Learning (ML) algorithms.

It comes with a set of powerful features:

- **Wide range of supported ML algorithms.** All supervised learning algorithms supported by `scikit-learn` are available, as well as Neural Networks (NNs) through PyTorch deep learning platform.
- **Built-in attack algorithms.** Evasion and poisoning attacks based on a custom-developed fast solver. In addition, we provide connectors to other third-party Adversarial Machine Learning libraries.
- **Dense/Sparse data support.** We provide full, transparent support for both dense (through `numpy` library) and sparse data (through `scipy` library) in a single data structure.
- **Visualize your results.** We provide visualization and plotting framework, based on the widely-known library matplotlib.
- **Explain your results.** Explainable ML methods to interpret model decisions via influential features and prototypes.
- **Model Zoo.** Use our pre-trained models to save time and easily replicate scientific results.
- **Multi-processing.** Do you want to save time further? We provide full compatibility with all the multi-processing features of `scikit-learn` and `pytorch`, along with built-in support of the joblib library.
- **Extensible.** Easily create new components, like ML models or attack algorithms, by extending the provided abstract interfaces.

CHAPTER

ONE

SOURCE CODE HOSTED AT: HTTPS://GITLAB.COM/SECML/SECML

SecML is currently in development. If you encounter any bugs, please report them using the GitLab issue tracker. Also, have a look at our ROADMAP for an overview of the future development directions.

1.1 Installation Guide

We recommend instaling SecML in a specific environment along with its dependencies.

Common frameworks to create and manage envs are virtualenv and conda. Both alternatives provide convenient user guides on how to properly setup the envs, so this guide will not cover the configuration procedure.

CHAPTER

TWO

OPERATING SYSTEM REQUIREMENTS

SecML can run under Python >= 3.5 with no additional configuration steps required, as all its dependencies are available as wheel packages for the primary macOS versions, Linux distributions and Windows.

However, to support additional advanced features more packages can be necessary depending on the Operating System used:

- Linux (Ubuntu 16.04 or later or equivalent distribution)
  - `python3-tk` for running MatplotLib Tk-based backends;
  - NVIDIA® CUDA® Toolkit for GPU support.
- macOS (10.12 Sierra or later)
  - Nothing to note.
- Windows (7 or later)
  - Microsoft Visual C++ Redistributable for Visual Studio 2015, 2017 and 2019.
  - NVIDIA® CUDA® Toolkit for GPU support.

CHAPTER

THREE

INSTALLATION PROCESS

Before starting the installation process try to obtain the latest version of the `pip` manager by calling: `pip install -U pip`

The setup process is managed by the Python package `setuptools`. Be sure to obtain the latest version by calling: `pip install -U setuptools`

Once the environment is set up, SecML can installed and run by multiple means:

1. Install from official PyPI repository:
    - `pip install secml`
2. Install from wheel/zip package (https://pypi.python.org/pypi/secml#files):
    - `pip install <package-file>`

In all cases, the setup process will try to install the correct dependencies. In case something goes wrong during the install process, try to install the dependencies **first** by calling: `pip install -r requirements.txt`

SecML should now be importable in python via: `import secml`.

To update a current installation using any of the previous methods, add the `-U` parameter after the `pip install` directive. Please see our Update Guides for specific upgrade intructions depending on the source and target version.

3.1 Extra Components

SecML comes with a set of extras components that can be installed if desired.

To specify the extra components to install, add the section `[extras]` while calling `pip install`. `extras` will be a comma-separated list of components you want to install. Example:

- `pip install secml[extra1,extra2]`

All the installation procedures via `pip` described above allow definition of the `[extras]` section.

CHAPTER

FOUR

AVAILABLE EXTRA COMPONENTS

- `pytorch` : Neural Networks (NNs) through PyTorch deep learning platform. Will install: `torch >= 1.1`, `torchvision >= 0.2.2` *Windows only*: the url to installation archives should be manually provided as `pip install secml[pytorch] -f https://download.pytorch.org/whl/torch_stable.html`.
- `cleverhans` : Wrapper of CleverHans, a Python library to benchmark vulnerability of machine learning systems to adversarial examples. Will install: `tensorflow >= 1.14.*, < 2`, `cleverhans`
- `tf-gpu` : Shortcut for installing `TensorFlow` package with GPU support (Linux and Windows only). Will install: `tensorflow-gpu >= 1.14.*, < 2`

4.1 Usage Guide

SecML is based on numpy, scipy, scikit-learn and pytorch, widely-used packages for scientific computing and machine learning with Python.

As a result, most of the interfaces of the library should be pretty familiar to frequent users of those packages.

The primary data class is the `secml.array.CArray`, multi-dimensional (currently limited to 2 dimensions) array structure which embeds both dense and sparse data accepting as input `numpy.ndarray` and `scipy.sparse.csr_matrix` (more sparse formats will be supported soon). This structure is the standard input and output of all other classes in the library.

The `secml.ml` package contains all the Machine Learning algorithms and support classes, including classifiers, loss and regularizer functions, kernels and performance evaluation functions. Also, a zoo of pre-trained models is provided by the `secml.model_zoo` package.

The `secml.adv` package contains evasion and poisoning attacks based on a custom-developed solver, along with classes to easily perform security evaluation of Machine Learning algorithms.

The `secml.explanation` package contains different explainable Machine Learning methods that allow interpreting classifiers decisions by analyzing the relevant components such as features or training prototypes.

The `secml.figure` package contains a visualization and plotting framework based on matplotlib.

4.2 Developers and Contributors

The contributing and developer's guide is available at: https://secml.gitlab.io/developers/

4.3 How to cite SecML

If you use SecML in a scientific publication, please cite the following paper:

secml: A Python Library for Secure and Explainable Machine Learning, Melis *et al.*, arXiv preprint arXiv:1912.10013 (2019).

BibTeX entry:

```
@article{melis2019secml,
  title={secml: A Python Library for Secure and Explainable Machine Learning},
  author={Melis, Marco and Demontis, Ambra and Pintor, Maura and Sotgiu, Angelo and
↪Biggio, Battista},
  journal={arXiv preprint arXiv:1912.10013},
  year={2019}
}
```

4.4 Authors

This library is maintained by PRALab - Pattern Recognition and Applications Lab.

List of contributors:

- Marco Melis [1]
- Ambra Demontis [1]
- Maura Pintor [1], [2]
- Battista Biggio [1], [2]

[1] Department of Electrical and Electronic Engineering, University of Cagliari, Italy [2] Pluribus One, Italy

4.5 Credits

- `numpy` Travis E, Oliphant. "A guide to NumPy", USA: Trelgol Publishing, 2006.
- `scipy` Travis E. Oliphant. "Python for Scientific Computing", Computing in Science & Engineering, 9, 10-20, 2007.
- `scikit-learn` Pedregosa et al., "Scikit-learn: Machine Learning in Python", JMLR 12, pp. 2825-2830, 2011.
- `matplotlib` J. D. Hunter, "Matplotlib: A 2D Graphics Environment", Computing in Science & Engineering, vol. 9, no. 3, pp. 90-95, 2007.
- `pytorch` Paszke, Adam, et al. "Automatic differentiation in pytorch.", NIPS-W, 2017.
- `cleverhans` Papernot, Nicolas, et al. "Technical Report on the CleverHans v2.1.0 Adversarial Examples Library." arXiv preprint arXiv:1610.00768 (2018).

4.6 Acknowledgements

SecML has been partially developed with the support of European Union's ALOHA project Horizon 2020 Research and Innovation programme, grant agreement No. 780788.

4.7 Copyright

4.7.1 Machine Learning

Training of Classifiers and Visualization of Results

In this first tutorial we aim to show some basic functionality of SecML.

Creation and visualization of a simple 2D dataset

The first step is loading the dataset. We are going to use a simple toy dataset consisting of 3 clusters of points, normally distributed.

Each dataset of SecML is a `CDataset` object, consisting of `dataset.X` and `dataset.Y`, where the samples and the corresponding labels are stored, respectively.

[1]:
```
random_state = 999

n_features = 2  # Number of features
n_samples = 1250  # Number of samples
centers = [[-2, 0], [2, -2], [2, 2]]  # Centers of the clusters
cluster_std = 0.8  # Standard deviation of the clusters

from secml.data.loader import CDLRandomBlobs
dataset = CDLRandomBlobs(n_features=n_features,
                         centers=centers,
                         cluster_std=cluster_std,
                         n_samples=n_samples,
                         random_state=random_state).load()
```

The dataset will be split in *training* and *test*, and normalized in the standard interval `[0, 1]` with a *min-max* normalizer.

[2]:
```
n_tr = 1000  # Number of training set samples
n_ts = 250  # Number of test set samples

# Split in training and test
from secml.data.splitter import CTrainTestSplit
splitter = CTrainTestSplit(
    train_size=n_tr, test_size=n_ts, random_state=random_state)
tr, ts = splitter.split(dataset)

# Normalize the data
from secml.ml.features import CNormalizerMinMax
```

(continues on next page)

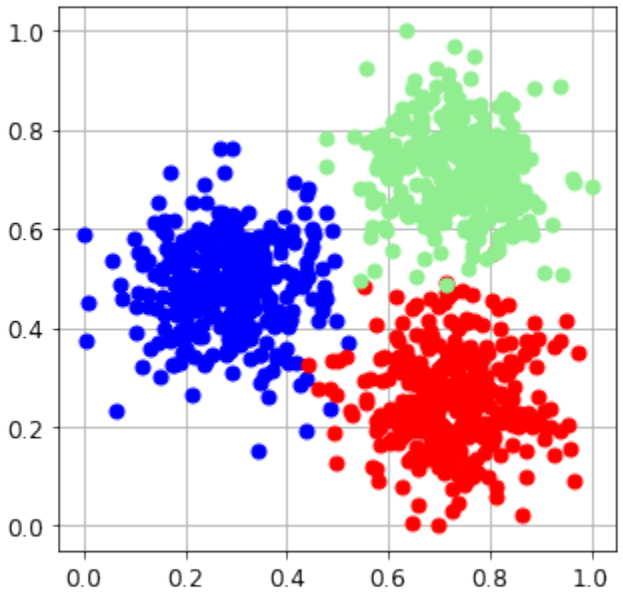
(continued from previous page)

```
nmz = CNormalizerMinMax()
tr.X = nmz.fit_transform(tr.X)
ts.X = nmz.transform(ts.X)
```

Let's visualize the dataset in a 2D plane.

The three clusters are clearly separable and normalized as we required.

[3]:
```
from secml.figure import CFigure
# Only required for visualization in notebooks
%matplotlib inline

fig = CFigure(width=5, height=5)

# Convenience function for plotting a dataset
fig.sp.plot_ds(tr)

fig.show()
```



Training of classifiers

Now we can train a **non-linear one-vs-all Support Vector Machine (SVM)**, using a **Radial Basis Function (RBF)** kernel for embedding.

We will evaluate the best training parameters through a *3-Fold Cross-Validation* procedure, using the accuracy as the performance metric. Each classifier has an integrated routine, `.estimate_parameters()` which estimates the best parameters on the given training set.

[4]:
```
# Creation of the multiclass classifier
from secml.ml.classifiers import CClassifierSVM
from secml.ml.kernels import CKernelRBF
svm = CClassifierSVM(kernel=CKernelRBF())
```

(continues on next page)

(continued from previous page)

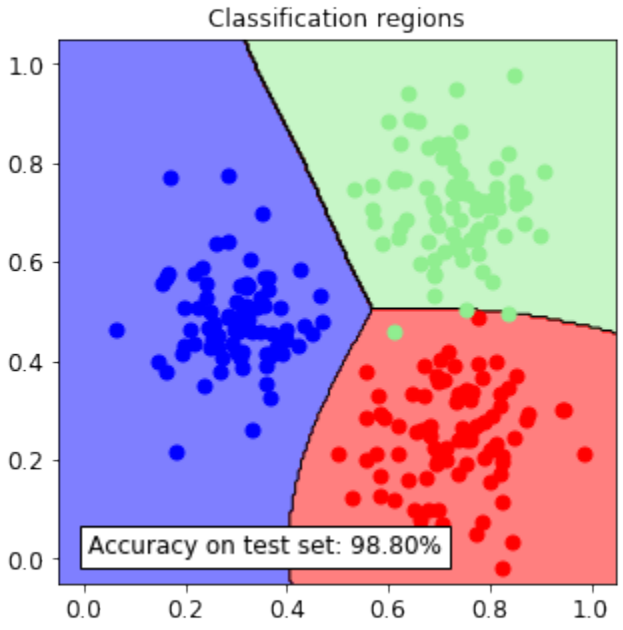
```
# Parameters for the Cross-Validation procedure
xval_params = {'C': [0.1, 1, 10], 'kernel.gamma': [1, 10, 100]}

# Let's create a 3-Fold data splitter
from secml.data.splitter import CDataSplitterKFold
xval_splitter = CDataSplitterKFold(num_folds=3, random_state=random_state)

# Metric to use for training and performance evaluation
from secml.ml.peval.metrics import CMetricAccuracy
metric = CMetricAccuracy()

# Select and set the best training parameters for the classifier
print("Estimating the best training parameters...")
best_params = svm.estimate_parameters(
    dataset=tr,
    parameters=xval_params,
    splitter=xval_splitter,
    metric=metric,
    perf_evaluator='xval'
)

print("The best training parameters are: ",
      [(k, best_params[k]) for k in sorted(best_params)])

# We can now fit the classifier
svm.fit(tr.X, tr.Y)

# Compute predictions on a test set
y_pred = svm.predict(ts.X)

# Evaluate the accuracy of the classifier
acc = metric.performance_score(y_true=ts.Y, y_pred=y_pred)

print("Accuracy on test set: {:.2%}".format(acc))
```

```
Estimating the best training parameters...
The best training parameters are:  [('C', 1), ('kernel.gamma', 10)]
Accuracy on test set: 98.80%
```

Visualization of the decision regions of the classifiers

Once the classifier is trained, we can visualize the *decision regions* over the entire feature space.

[6]:
```
fig = CFigure(width=5, height=5)

# Convenience function for plotting the decision function of a classifier
fig.sp.plot_decision_regions(svm, n_grid_points=200)

fig.sp.plot_ds(ts)
fig.sp.grid(grid_on=False)

fig.sp.title("Classification regions")
fig.sp.text(0.01, 0.01, "Accuracy on test set: {:.2%}".format(acc),
            bbox=dict(facecolor='white'))
fig.show()
```



Training other classifiers

Now we can repeat the above process for other classifiers available in SecML. We are going to use a `namedtuple` for easy storage of objects and parameters.

Binary classifiers like `CClassifierSGD` can be extended to multiclass one-vs-all schemes using `CMulticlassClassifierOVA`.

Please note that parameters estimation may take a while (up to a few minutes) depending on the machine the script is run on.

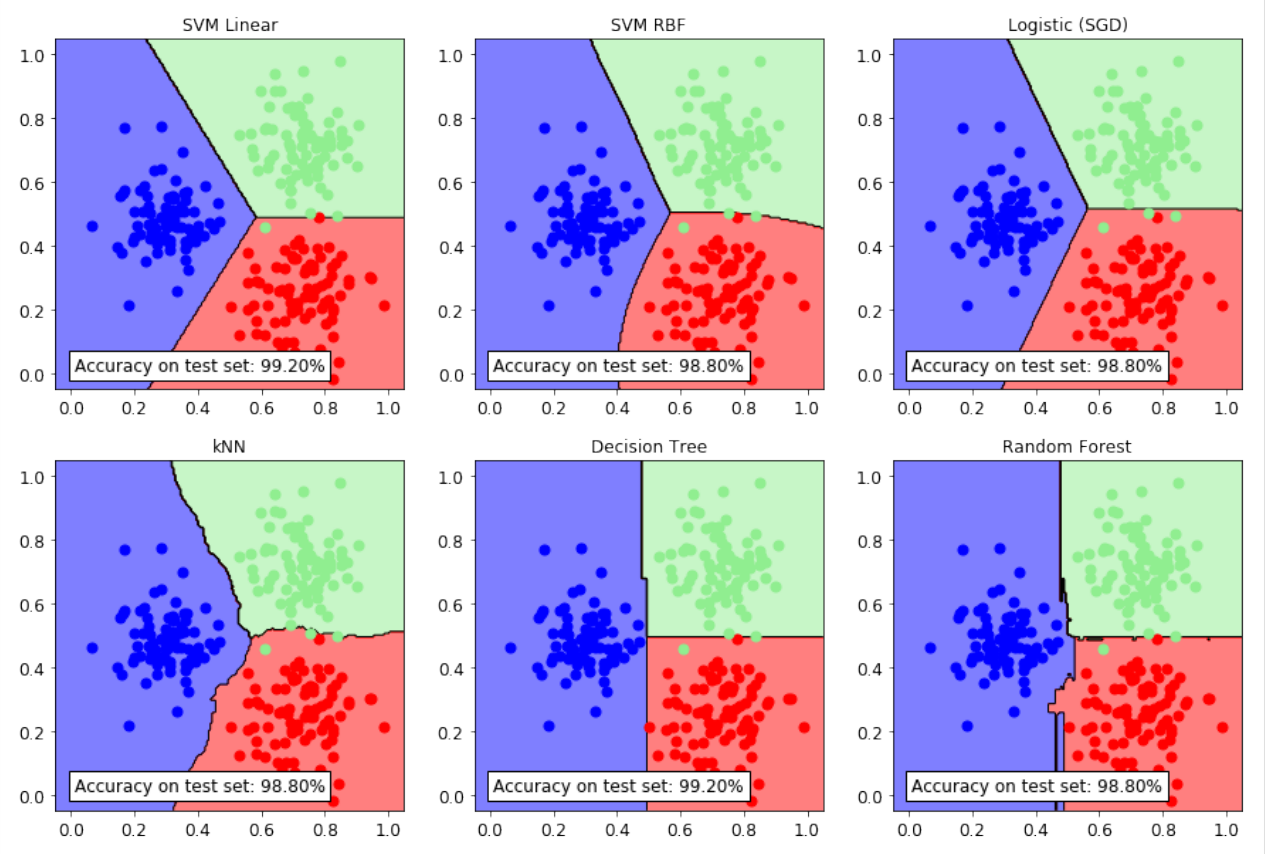
[9]:
```
from collections import namedtuple
CLF = namedtuple('CLF', 'clf_name clf xval_parameters')

# Binary classifiers
from secml.ml.classifiers import CClassifierSGD
from secml.ml.classifiers.multiclass import CClassifierMulticlassOVA
# Natively-multiclass classifiers
from secml.ml.classifiers import CClassifierSVM, CClassifierKNN,␣
↪CClassifierDecisionTree, CClassifierRandomForest

clf_list = [
    CLF(
        clf_name='SVM Linear',
        clf=CClassifierSVM(),
        xval_parameters={'C': [0.1, 1, 10]}),
    CLF(clf_name='SVM RBF',
        clf=CClassifierSVM(kernel='rbf'),
        xval_parameters={'C': [0.1, 1, 10],  'kernel.gamma': [1, 10, 100]}),
    CLF(clf_name='Logistic (SGD)',
        clf=CClassifierMulticlassOVA(
            CClassifierSGD, regularizer='l2', loss='log',
```

(continues on next page)

(continued from previous page)

```
            random_state=random_state),
        xval_parameters={'alpha': [1e-7, 1e-6, 1e-5]}),
    CLF(clf_name='kNN',
        clf=CClassifierKNN(),
        xval_parameters={'n_neighbors': [5, 10, 20]}),
    CLF(clf_name='Decision Tree',
        clf=CClassifierDecisionTree(random_state=random_state),
        xval_parameters={'max_depth': [1, 3, 5]}),
    CLF(clf_name='Random Forest',
        clf=CClassifierRandomForest(random_state=random_state),
        xval_parameters={'n_estimators': [10, 20, 30]}),
]

from secml.data.splitter import CDataSplitterKFold
xval_splitter = CDataSplitterKFold(num_folds=3, random_state=random_state)

fig = CFigure(width=5 * len(clf_list) / 2, height=5 * 2)

for i, test_case in enumerate(clf_list):

    clf = test_case.clf
    xval_params = test_case.xval_parameters

    print("\nEstimating the best training parameters of {:} ..."
          "".format(test_case.clf_name))

    best_params = clf.estimate_parameters(
        dataset=tr, parameters=xval_params, splitter=xval_splitter,
        metric='accuracy', perf_evaluator='xval')

    print("The best parameters for '{:}' are: ".format(test_case.clf_name),
          [(k, best_params[k]) for k in sorted(best_params)])

    print("Training of {:} ...".format(test_case.clf_name))
    clf.fit(tr.X, tr.Y)

    # Predictions on test set and performance evaluation
    y_pred = clf.predict(ts.X)
    acc = metric.performance_score(y_true=ts.Y, y_pred=y_pred)

    print("Classifier: {:}\tAccuracy: {:.2%}".format(test_case.clf_name, acc))

    # Plot the decision function
    from math import ceil
    # Use `CFigure.subplot` to divide the figure in multiple subplots
    fig.subplot(2, int(ceil(len(clf_list) / 2)), i + 1)

    fig.sp.plot_decision_regions(clf, n_grid_points=200)

    fig.sp.plot_ds(ts)
    fig.sp.grid(grid_on=False)

    fig.sp.title(test_case.clf_name)
    fig.sp.text(0.01, 0.01, "Accuracy on test set: {:.2%}".format(acc),
                bbox=dict(facecolor='white'))

fig.show()
```

```
Estimating the best training parameters of SVM Linear ...
The best parameters for 'SVM Linear' are:  [('C', 1)]
Training of SVM Linear ...
Classifier: SVM Linear  Accuracy: 99.20%

Estimating the best training parameters of SVM RBF ...
The best parameters for 'SVM RBF' are:  [('C', 1), ('kernel.gamma', 10)]
Training of SVM RBF ...
Classifier: SVM RBF     Accuracy: 98.80%

Estimating the best training parameters of Logistic (SGD) ...
The best parameters for 'Logistic (SGD)' are:  [('alpha', 1e-06)]
Training of Logistic (SGD) ...
Classifier: Logistic (SGD)      Accuracy: 98.80%

Estimating the best training parameters of kNN ...
The best parameters for 'kNN' are:  [('n_neighbors', 10)]
Training of kNN ...
Classifier: kNN Accuracy: 98.80%

Estimating the best training parameters of Decision Tree ...
The best parameters for 'Decision Tree' are:  [('max_depth', 3)]
Training of Decision Tree ...
Classifier: Decision Tree       Accuracy: 99.20%

Estimating the best training parameters of Random Forest ...
The best parameters for 'Random Forest' are:  [('n_estimators', 20)]
Training of Random Forest ...
Classifier: Random Forest       Accuracy: 98.80%
```



[ ]:

Neural Networks with PyTorch

In this tutorial we will show how to create a Neural Network using the PyTorch (more usage examples of PyTorch here).

Warning

Requires installation of the `pytorch` extra dependency. See *extra components* for more information.

Classifying blobs

First, we need to create a neural network. We simply use PyTorch `nn.Module` as regular PyTorch code.

[1]:
```
import torch
from torch import nn

class Net(nn.Module):
    """
    Model with input size (-1, 5) for blobs dataset
    with 5 features
    """
```

(continues on next page)

(continued from previous page)

```
    def __init__(self, n_features, n_classes):
        """Example network."""
        super(Net, self).__init__()
        self.fc1 = nn.Linear(n_features, 5)
        self.fc2 = nn.Linear(5, n_classes)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

We will use a 5D dataset composed with 3 gaussians. We can use 4k samples for training and 1k for testing. We can divide the sets in batches so that they can be processed in small groups by the network. We use a batch size of 20.

[2]:
```
# experiment parameters
n_classes = 3
n_features = 2
n_samples_tr = 4000  # number of training set samples
n_samples_ts = 1000  # number of testing set samples
batch_size = 20

# dataset creation
from secml.data.loader import CDLRandom
dataset = CDLRandom(n_samples=n_samples_tr + n_samples_ts,
                    n_classes=n_classes,
                    n_features=n_features, n_redundant=0,
                    n_clusters_per_class=1,
                    class_sep=1, random_state=0).load()

# Split in training and test
from secml.data.splitter import CTrainTestSplit
splitter = CTrainTestSplit(train_size=n_samples_tr,
                           test_size=n_samples_ts,
                           random_state=0)

tr, ts = splitter.split(dataset)

# Normalize the data
from secml.ml.features.normalization import CNormalizerMinMax
nmz = CNormalizerMinMax()
tr.X = nmz.fit_transform(tr.X)
ts.X = nmz.transform(ts.X)
```

Now we can create an instance of the PyTorch model and then wrap it in the specific class that will link it to our library functionalities.

[3]:
```
# Random seed
torch.manual_seed(0)

# torch model creation
net = Net(n_features=n_features, n_classes=n_classes)

from torch import optim
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(),
                      lr=0.001, momentum=0.9)
```

(continues on next page)

(continued from previous page)

```
# wrap torch model in CClassifierPyTorch class
from secml.ml.classifiers import CClassifierPyTorch
clf = CClassifierPyTorch(model=net,
                         loss=criterion,
                         optimizer=optimizer,
                         input_shape=(n_features,),
                         random_state=0)
```

We can simply use the loaded `CDataset` and pass it to the `fit` method. The wrapper will handle batch processing and train the network for the number of epochs specified in the wrapper constructor.

[4]:
```
# clf.verbose = 1  # Can be used to display training process output
print("Training started...")
clf.fit(tr.X, tr.Y)
print("Training completed!")
```

```
Training started...
Training completed!
```

Using the model in "predict" mode is just as easy. We can use the method `predict` defined in our wrapper, and pass in the data. We can evaluate the accuracy with the `CMetric` defined in our library.

[5]:
```
label_torch = clf.predict(ts.X, return_decision_function=False)

from secml.ml.peval.metrics import CMetric
acc_torch = CMetric.create('accuracy').performance_score(ts.Y, label_torch)

print("Model Accuracy: {}".format(acc_torch))
```

```
Model Accuracy: 0.991
```

4.7.2 Adversarial Machine Learning

Evasion Attacks against Machine Learning

In this tutorial we will experiment with **adversarial evasion attacks** against a Support Vector Machine (SVM) with the Radial Basis Function (RBF) kernel.

Evasion attacks (a.k.a. *adversarial examples*) consists of carefully perturbing the input samples at *test time* to have them misclassified.

We will first create and train the classifier, evaluating its performance in the standard scenario, *i.e. not under attack*.

The following part replicates the procedure from the *first tutorial*.

[1]:
```
random_state = 999

n_features = 2  # Number of features
n_samples = 1100  # Number of samples
centers = [[-2, 0], [2, -2], [2, 2]]  # Centers of the clusters
cluster_std = 0.8  # Standard deviation of the clusters

from secml.data.loader import CDLRandomBlobs
dataset = CDLRandomBlobs(n_features=n_features,
                         centers=centers,
```

(continues on next page)

(continued from previous page)

```
                          cluster_std=cluster_std,
                          n_samples=n_samples,
                          random_state=random_state).load()

n_tr = 1000  # Number of training set samples
n_ts = 100  # Number of test set samples

# Split in training and test
from secml.data.splitter import CTrainTestSplit
splitter = CTrainTestSplit(
    train_size=n_tr, test_size=n_ts, random_state=random_state)
tr, ts = splitter.split(dataset)

# Normalize the data
from secml.ml.features import CNormalizerMinMax
nmz = CNormalizerMinMax()
tr.X = nmz.fit_transform(tr.X)
ts.X = nmz.transform(ts.X)

# Metric to use for training and performance evaluation
from secml.ml.peval.metrics import CMetricAccuracy
metric = CMetricAccuracy()

# Creation of the multiclass classifier
from secml.ml.classifiers import CClassifierSVM
from secml.ml.classifiers.multiclass import CClassifierMulticlassOVA
from secml.ml.kernels import CKernelRBF
clf = CClassifierMulticlassOVA(CClassifierSVM, kernel=CKernelRBF())

# Parameters for the Cross-Validation procedure
xval_params = {'C': [1e-2, 0.1, 1], 'kernel.gamma': [10, 100, 1e3]}

# Let's create a 3-Fold data splitter
from secml.data.splitter import CDataSplitterKFold
xval_splitter = CDataSplitterKFold(num_folds=3, random_state=random_state)

# Select and set the best training parameters for the classifier
print("Estimating the best training parameters...")
best_params = clf.estimate_parameters(
    dataset=tr,
    parameters=xval_params,
    splitter=xval_splitter,
    metric='accuracy',
    perf_evaluator='xval'
)

print("The best training parameters are: ",
      [(k, best_params[k]) for k in sorted(best_params)])

# We can now fit the classifier
clf.fit(tr.X, tr.Y)

# Compute predictions on a test set
y_pred = clf.predict(ts.X)

# Evaluate the accuracy of the classifier
acc = metric.performance_score(y_true=ts.Y, y_pred=y_pred)
```

(continues on next page)

(continued from previous page)

```
print("Accuracy on test set: {:.2%}".format(acc))
```

```
Estimating the best training parameters...
The best training parameters are:  [('C', 0.1), ('kernel.gamma', 100)]
Accuracy on test set: 99.00%
```

Crafting Adversarial Examples

We are going to create an adversarial example against the SVM classifier using the **gradient-based maximum-confidence** algorithm for generating evasion attacks proposed in:

> [biggio13-ecml] Biggio, B., Corona, I., Maiorca, D., Nelson, B., Šrndić, N., Laskov, P., Giacinto, G., Roli, F., 2013. Evasion Attacks against Machine Learning at Test Time. In ECML-PKDD 2013.
>
> [melis17-vipar] Melis, M., Demontis, A., Biggio, B., Brown, G., Fumera, G. and Roli, F., 2017. Is deep learning safe for robot vision? adversarial examples against the icub humanoid. In Proceedings of IEEE ICCV 2017.
>
> [demontis19-usenix] Demontis, A., Melis, M., Pintor, M., Jagielski, M., Biggio, B., Oprea, A., Nita-Rotaru, C. and Roli, F., 2019. Why Do Adversarial Attacks Transfer? Explaining Transferability of Evasion and Poisoning Attacks. In 28th Usenix Security Symposium, Santa Clara, California, USA.

which is implemented in SecML by the `CAttackEvasionPGDLS` class.

Let's define the attack parameters. First, we choose to generate an *l2* perturbation within a maximum ball of radius `eps = 0.4` from the initial point. The maximum perturbation value is denoted as `dmax` in our implementation. Second, we also add a low/upper bound as our feature space is limited in `[0, 1]`. Last, as we are not interested in generating an adversarial example for a specific class, we perform an error-generic attack by setting `y_target = None`.

The attack internally uses a solver based on *Projected Gradient Descent with Bisect Line Search*, implemented by the `COptimizerPGDLS` class. The parameters of the solver can be specified while creating the attack and must be optimized depending on the specific optimization problem.

[2]:
```
x0, y0 = ts[5, :].X, ts[5, :].Y  # Initial sample

noise_type = 'l2'  # Type of perturbation 'l1' or 'l2'
dmax = 0.4  # Maximum perturbation
lb, ub = 0, 1  # Bounds of the attack space. Can be set to `None` for unbounded
y_target = None  # None if `error-generic` or a class label for `error-specific`

# Should be chosen depending on the optimization problem
solver_params = {
    'eta': 0.3,
    'eta_min': 0.1,
    'eta_max': None,
    'max_iter': 100,
    'eps': 1e-4
}

from secml.adv.attacks.evasion import CAttackEvasionPGDLS
pgd_ls_attack = CAttackEvasionPGDLS(
    classifier=clf,
    double_init_ds=tr,
    double_init=False,
```

(continues on next page)

(continued from previous page)

```
    distance=noise_type,
    dmax=dmax,
    lb=lb, ub=ub,
    solver_params=solver_params,
    y_target=y_target)

# Run the evasion attack on x0
y_pred_pgdls, _, adv_ds_pgdls, _ = pgd_ls_attack.run(x0, y0)

print("Original x0 label: ", y0.item())
print("Adversarial example label (PGD-LS): ", y_pred_pgdls.item())

print("Number of classifier gradient evaluations: {:}"
      "".format(pgd_ls_attack.grad_eval))
```

```
Original x0 label:  1
Adversarial example label (PGD-LS):  2
Number of classifier gradient evaluations: 7
```

Let's now test another attack algorithm, implemented by `CAttackEvasionPGD`, which leverages the standard *Projected Gradient Descent* solver. We keep the same attack parameters as before.

[3]:
```
# Should be chosen depending on the optimization problem
solver_params = {
    'eta': 0.3,
    'max_iter': 100,
    'eps': 1e-4
}

from secml.adv.attacks.evasion import CAttackEvasionPGD
pgd_attack = CAttackEvasionPGD(
    classifier=clf,
    double_init_ds=tr,
    double_init=False,
    distance=noise_type,
    dmax=dmax,
    lb=lb, ub=ub,
    solver_params=solver_params,
    y_target=y_target)

# Run the evasion attack on x0
y_pred_pgd, _, adv_ds_pgd, _ = pgd_attack.run(x0, y0)

print("Original x0 label: ", y0.item())
print("Adversarial example label (PGD): ", y_pred_pgd.item())

print("Number of classifier gradient evaluations: {:}"
      "".format(pgd_attack.grad_eval))
```

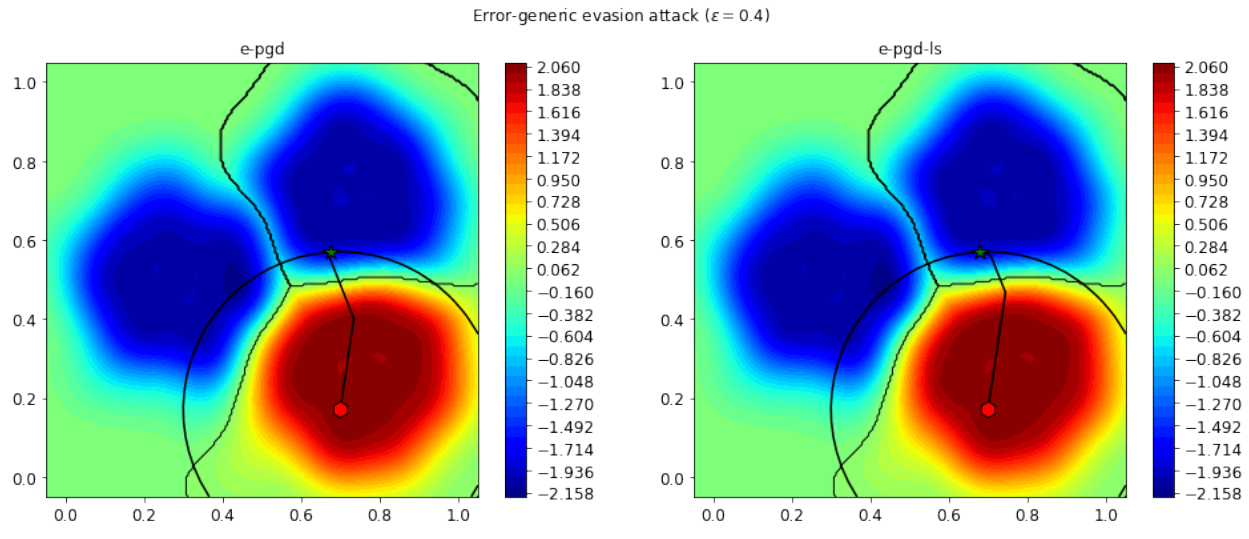
```
Original x0 label:  1
Adversarial example label (PGD):  2
Number of classifier gradient evaluations: 40
```

We can see that the classifier has been successfully evaded in both cases. However, the `PGD-LS` solver with bisect line search queries the classifier gradient function many times less, thus generating the adversarial examples much faster.

Let's now visualize both the attacks on a 2D plane. On the background, the value of the objective function of the

attacks is shown.

[4]:
```
from secml.figure import CFigure
# Only required for visualization in notebooks
%matplotlib inline

fig = CFigure(width=16, height=6, markersize=12)

# Let's replicate the `l2` constraint used by the attack for visualization
from secml.optim.constraints import CConstraintL2
constraint = CConstraintL2(center=x0, radius=dmax)

for i, (attack, adv_ds) in enumerate(
        [(pgd_attack, adv_ds_pgd), (pgd_ls_attack, adv_ds_pgdls)]):

    fig.subplot(1, 2, i + 1)

    # Convenience function for plotting the attack objective function
    fig.sp.plot_fun(attack.objective_function, plot_levels=False,
                    multipoint=True, n_grid_points=200)
    # Let's also plot the decision boundaries of the classifier
    fig.sp.plot_decision_regions(clf, plot_background=False, n_grid_points=200)

    # Construct an array with the original point and the adversarial example
    adv_path = x0.append(adv_ds.X, axis=0)

    # Convenience function for plotting the optimization sequence
    fig.sp.plot_path(attack.x_seq)

    # Convenience function for plotting a constraint
    fig.sp.plot_constraint(constraint)

    fig.sp.title(attack.class_type)
    fig.sp.grid(grid_on=False)

fig.title(r"Error-generic evasion attack ($\varepsilon={:}$)".format(dmax))

fig.show()
```



We can see that the initial point **x0** (red hexagon) has been perturbed in the feature space so that is actually classified by

the SVM as a point from another class. The final *adversarial example* is the green star. We also show the *l2* constraint as a black circle which has limited the maximum perturbation applicable to **x0**.

Security evaluation of a classifier

We could be interested in evaluating the **robustness** of a classifier against increasing values of the maximum perturbation `eps`.

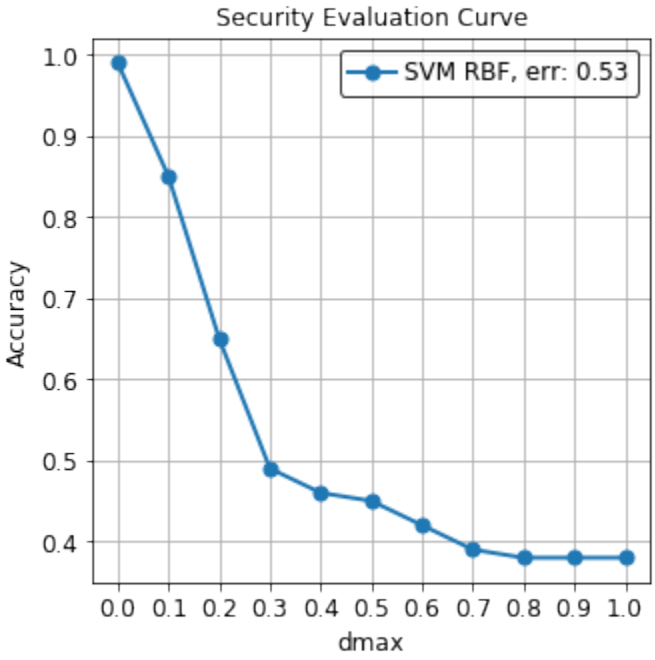
SecML provides a way to easily produce a **Security Evaluation Curve**, by means of the `CSecEval` class.

The `CSecEval` instance will take a `CAttack` as input and will test the classifier using the desired perturbation levels.

Please note that the security evaluation process may take a while (up to a few minutes) depending on the machine the script is run on.

[5]:
```
# Perturbation levels to test
from secml.array import CArray
e_vals = CArray.arange(start=0, step=0.1, stop=1.1)

from secml.adv.seceval import CSecEval
sec_eval = CSecEval(
    attack=pgd_ls_attack, param_name='dmax', param_values=e_vals)

# Run the security evaluation using the test set
print("Running security evaluation...")
sec_eval.run_sec_eval(ts)

from secml.figure import CFigure
fig = CFigure(height=5, width=5)

# Convenience function for plotting the Security Evaluation Curve
fig.sp.plot_sec_eval(
    sec_eval.sec_eval_data, marker='o', label='SVM RBF', show_average=True)
```

```
Running security evaluation...
```



We can see how the SVM classifier is *vulnerable* to adversarial attacks and we are able to evade it even with small perturbations.

For further reference about the security evaluation of machine-learning models under attack see:

> [biggio13-tkde] Biggio, B., Fumera, G. and Roli, F., 2013. Security evaluation of pattern classifiers under attack. In IEEE transactions on knowledge and data engineering.
>
> [biggio18-pr] Biggio, B. and Roli, F., 2018. Wild patterns: Ten years after the rise of adversarial machine learning. In Pattern Recognition.

Transferability of Evasion Attacks

Transferability captures the ability of an attack against a machine-learning model to be effective against a different, potentially unknown, model.

In this tutorial we are going to test if an evasion attack generated against a Support Vector Machine (SVM), the *surrogate* classifier, will transfer to other classifiers, the *targets*, or not.

For more details about the transferability property of adversarial attacks please refer to:

> [demontis19-usenix] Demontis, A., Melis, M., Pintor, M., Jagielski, M., Biggio, B., Oprea, A., Nita-Rotaru, C. and Roli, F., 2019. Why Do Adversarial Attacks Transfer? Explaining Transferability of Evasion and Poisoning Attacks. In 28th Usenix Security Symposium, Santa Clara, California, USA.

We will first create and train the surrogate and different target classifiers, evaluating their performance in the standard scenario, *i.e. not under attack*. The surrogate and the target classifiers will be trained on different training sets.

The following part partially replicates the procedure from the *first tutorial*.

[1]:
```
random_state = 999

n_features = 2  # Number of features
n_samples = 2250  # Number of samples
```

(continues on next page)

(continued from previous page)

```
centers = [[-2, 0], [2, -2], [2, 2]]  # Centers of the clusters
cluster_std = 0.8  # Standard deviation of the clusters

from secml.data.loader import CDLRandomBlobs
dataset = CDLRandomBlobs(n_features=n_features,
                         centers=centers,
                         cluster_std=cluster_std,
                         n_samples=n_samples,
                         random_state=random_state).load()

n_tr = 1000  # Number of training set samples
n_ts = 250  # Number of test set samples

# Split in training and test.
from secml.data.splitter import CTrainTestSplit
splitter = CTrainTestSplit(
    train_size=2 * n_tr, test_size=n_ts, random_state=random_state)
tr, ts = splitter.split(dataset)

# Normalize the data
from secml.ml.features import CNormalizerMinMax
nmz = CNormalizerMinMax()
tr.X = nmz.fit_transform(tr.X)
ts.X = nmz.transform(ts.X)

# Generate 2 training datasets for surrogate and target classifiers
tr1 = tr[:n_tr, :]  # Source classifier training set
tr2 = tr[n_tr:, :]  # Target classifier training set

from collections import namedtuple
CLF = namedtuple('CLF', 'clf_name clf xval_parameters')

from secml.ml.classifiers.multiclass import CClassifierMulticlassOVA
# Binary classifiers
from secml.ml.classifiers import CClassifierSVM, CClassifierSGD
# Natively-multiclass classifiers
from secml.ml.classifiers import CClassifierKNN, CClassifierDecisionTree,␣
↪CClassifierRandomForest

# Let's create a 3-Fold data splitter
from secml.data.splitter import CDataSplitterKFold
xval_splitter = CDataSplitterKFold(num_folds=3, random_state=random_state)

# Metric to use for training and performance evaluation
from secml.ml.peval.metrics import CMetricAccuracy
metric = CMetricAccuracy()

surr_clf = CLF(
    clf_name='SVM Linear',
    clf=CClassifierMulticlassOVA(CClassifierSVM, kernel='linear'),
    xval_parameters={'C': [1e-2, 0.1, 1]})

print("Estimating the best training parameters of the surrogate classifier...")
best_params = surr_clf.clf.estimate_parameters(
    dataset=tr1,
    parameters=surr_clf.xval_parameters,
    splitter=xval_splitter,
```

(continues on next page)

(continued from previous page)

```
    metric=metric,
    perf_evaluator='xval'
)

print("The best training parameters of the surrogate classifier are: ",
      [(k, best_params[k]) for k in sorted(best_params)])

surr_clf.clf.fit(tr1.X, tr1.Y)

y_pred = surr_clf.clf.predict(ts.X)

acc = metric.performance_score(y_true=ts.Y, y_pred=y_pred)

print("Accuracy of the surrogate classifier on test set: {:.2%}".format(acc))

print("\nTraining the target classifiers...")

target_clf_list = [
    CLF(
        clf_name='SVM Linear',
        clf=CClassifierMulticlassOVA(CClassifierSVM, kernel='linear'),
        xval_parameters={'C': [1e-2, 0.1, 1]}),
    CLF(clf_name='SVM RBF',
        clf=CClassifierMulticlassOVA(CClassifierSVM, kernel='rbf'),
        xval_parameters={'C': [1e-2, 0.1, 1],  'kernel.gamma': [1, 10, 100]}),
    CLF(clf_name='Logistic (SGD)',
        clf=CClassifierMulticlassOVA(
            CClassifierSGD, regularizer='l2', loss='log',
            random_state=random_state),
        xval_parameters={'alpha': [1e-6, 1e-5, 1e-4]}),
    CLF(clf_name='kNN',
        clf=CClassifierKNN(),
        xval_parameters={'n_neighbors': [30, 40, 50]}),
    CLF(clf_name='Decision Tree',
        clf=CClassifierDecisionTree(random_state=random_state),
        xval_parameters={'max_depth': [1, 3, 5]}),
    CLF(clf_name='Random Forest',
        clf=CClassifierRandomForest(random_state=random_state),
        xval_parameters={'n_estimators': [20, 30, 40]}),
]

for i, test_case in enumerate(target_clf_list):

    clf = test_case.clf
    xval_params = test_case.xval_parameters

    print("\nEstimating the best training parameters of {:} ..."
          "".format(test_case.clf_name))

    best_params = clf.estimate_parameters(
        dataset=tr2, parameters=xval_params, splitter=xval_splitter,
        metric='accuracy', perf_evaluator='xval')

    print("The best parameters for '{:}' are: ".format(test_case.clf_name),
          [(k, best_params[k]) for k in sorted(best_params)])

    print("Training of {:} ...".format(test_case.clf_name))
```

(continues on next page)

(continued from previous page)

```
    clf.fit(tr2.X, tr2.Y)

    # Predictions on test set and performance evaluation
    y_pred = clf.predict(ts.X)
    acc = metric.performance_score(y_true=ts.Y, y_pred=y_pred)

    print("Classifier: {:}\tAccuracy: {:.2%}".format(test_case.clf_name, acc))
```

```
Estimating the best training parameters of the surrogate classifier...
The best training parameters of the surrogate classifier are:  [('C', 0.1)]
Accuracy of the surrogate classifier on test set: 99.60%

Training the target classifiers...

Estimating the best training parameters of SVM Linear ...
The best parameters for 'SVM Linear' are:  [('C', 0.1)]
Training of SVM Linear ...
Classifier: SVM Linear  Accuracy: 99.60%

Estimating the best training parameters of SVM RBF ...
The best parameters for 'SVM RBF' are:  [('C', 0.01), ('kernel.gamma', 10)]
Training of SVM RBF ...
Classifier: SVM RBF     Accuracy: 99.60%

Estimating the best training parameters of Logistic (SGD) ...
The best parameters for 'Logistic (SGD)' are:  [('alpha', 1e-06)]
Training of Logistic (SGD) ...
Classifier: Logistic (SGD)      Accuracy: 99.60%

Estimating the best training parameters of kNN ...
The best parameters for 'kNN' are:  [('n_neighbors', 30)]
Training of kNN ...
Classifier: kNN Accuracy: 99.60%

Estimating the best training parameters of Decision Tree ...
The best parameters for 'Decision Tree' are:  [('max_depth', 3)]
Training of Decision Tree ...
Classifier: Decision Tree       Accuracy: 98.80%

Estimating the best training parameters of Random Forest ...
The best parameters for 'Random Forest' are:  [('n_estimators', 30)]
Training of Random Forest ...
Classifier: Random Forest       Accuracy: 98.80%
```

Generation of the Adversarial Examples

As done in the *evasion tutorial*, we now craft the adversarial examples using the **gradient-based maximum-confidence** algorithm for generating evasion attacks, implemented by the `CAttackEvasionPGDLS` class (`e-pgd-ls`).

This time, we are going to generate an **error-specific** attack by setting `y_target` to one of the classes of the dataset. In this way, we enforce the solver to perturb the points so that the classifier will classify them with the `y_true` label.

Please note that the attack on multiple samples may take a while (up to a few minutes) depending on the machine the script is run on.

[2]:
```
noise_type = 'l2'  # Type of perturbation 'l1' or 'l2'
dmax = 0.4  # Maximum perturbation
lb, ub = 0, 1  # Bounds of the attack space. Can be set to `None` for unbounded
y_target = 2  # `error-specific` attack. None for `error-generic`

# Should be chosen depending on the optimization problem
solver_params = {
    'eta': 1e-1,
    'eta_min': 0.1,
    'eta_max': None,
    'max_iter': 100,
    'eps': 1e-4
}

from secml.adv.attacks.evasion import CAttackEvasionPGDLS
pgd_ls_attack = CAttackEvasionPGDLS(
    classifier=surr_clf.clf,
    double_init_ds=tr1,
    double_init=False,
    distance=noise_type,
    dmax=dmax,
    lb=lb, ub=ub,
    solver_params=solver_params,
    y_target=y_target)

# Run the evasion attack on x0
print("Attack started...")
y_pred, scores, adv_ds, f_obj = pgd_ls_attack.run(ts.X, ts.Y)
print("Attack complete!")
```

```
Attack started...
Attack complete!
```

Analysis of Transferability

Let's now test if the previously generated examples transfer to other models.

Initally, we test the performance of each target classifier on the adversarial examples. Later, we plot few of the samples on a 2D plane.

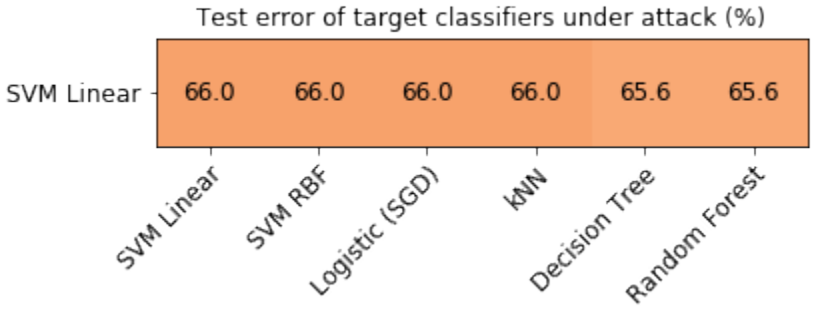
[3]:
```
# Metric to use for testing transferability
from secml.ml.peval.metrics import CMetricTestError
metric = CMetricTestError()

trans_error = []
transfer_rate = 0.0
for target_clf in target_clf_list:

    print("\nTesting transferability of {:}".format(target_clf.clf_name))

    origin_error = metric.performance_score(
            y_true=ts.Y, y_pred=target_clf.clf.predict(ts.X))

    print("Test error (no attack): {:.2%}".format(origin_error))

    trans_error_clf = metric.performance_score(
```

(continues on next page)

(continued from previous page)

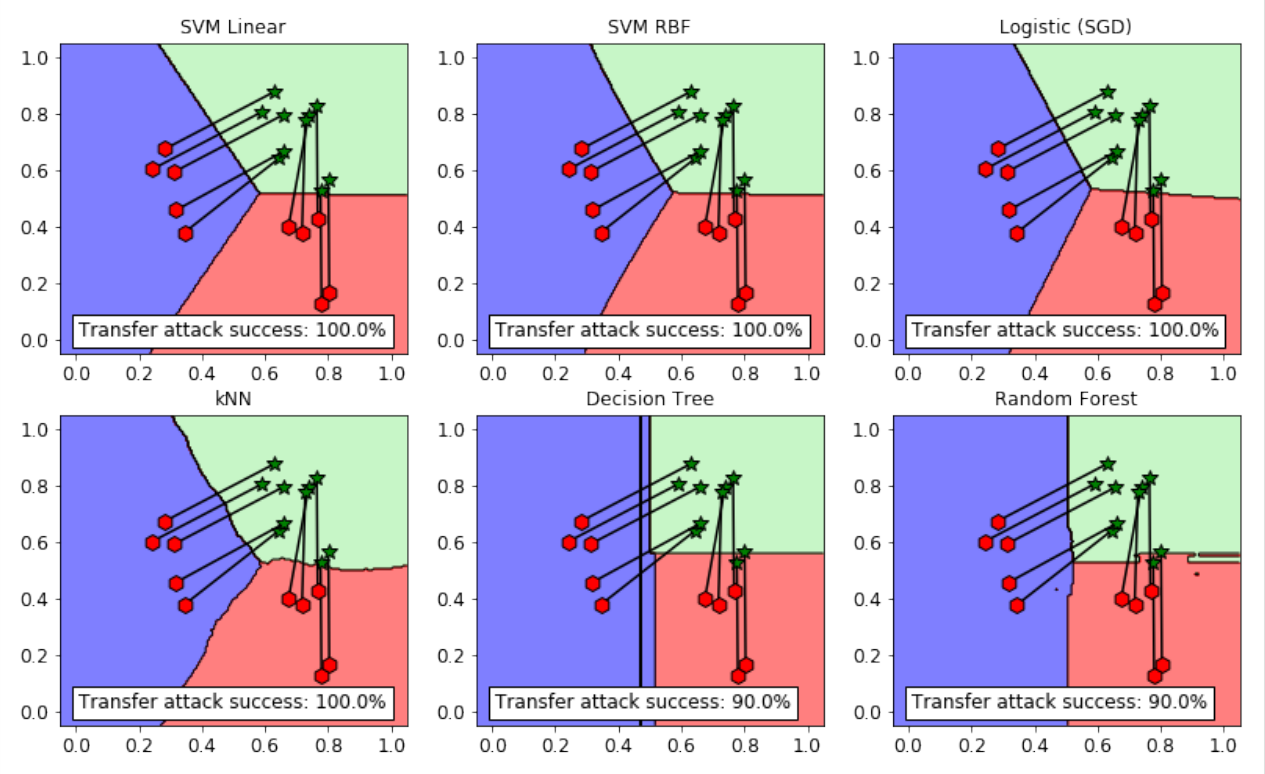
```
        y_true=ts.Y, y_pred=target_clf.clf.predict(adv_ds.X))

    trans_error.append(trans_error_clf)
    transfer_rate += trans_error_clf

# Computing the transfer rate
transfer_rate /= len(target_clf_list)

from secml.array import CArray
trans_acc = CArray(trans_error) * 100  # Show results in percentage

from secml.figure import CFigure
# Only required for visualization in notebooks
%matplotlib inline

fig = CFigure(height=1)
a = fig.sp.imshow(trans_acc.reshape((1, 6)),
                  cmap='Oranges', interpolation='nearest',
                  alpha=.65, vmin=60, vmax=70)

fig.sp.xticks(CArray.arange((len(target_clf_list))))
fig.sp.xticklabels([c.clf_name for c in target_clf_list],
                   rotation=45, ha="right", rotation_mode="anchor")
fig.sp.yticks([0])
fig.sp.yticklabels([surr_clf.clf_name])

for i in range(len(target_clf_list)):
    fig.sp.text(i, 0, trans_acc[i].round(2).item(), va='center', ha='center')

fig.sp.title("Test error of target classifiers under attack (%)")

fig.show()

print("\nAverage transfer rate: {:.2%}".format(transfer_rate))
```

```
Testing transferability of SVM Linear
Test error (no attack): 0.40%

Testing transferability of SVM RBF
Test error (no attack): 0.40%

Testing transferability of Logistic (SGD)
Test error (no attack): 0.40%

Testing transferability of kNN
Test error (no attack): 0.40%

Testing transferability of Decision Tree
Test error (no attack): 1.20%

Testing transferability of Random Forest
Test error (no attack): 1.20%

Average transfer rate: 65.87%
```



We can observe how the accuracy of the target classifiers on the adversarial point generated against the surrogate classifier is extremely low, which highlights how the machine-learning models are **vulnerable to transfer attacks**.

[4]:
```
from secml.figure import CFigure
from secml.array import CArray
from math import ceil
fig = CFigure(width=4.5 * len(target_clf_list) / 2,
              height=4 * 2, markersize=10)

for clf_idx in range(len(target_clf_list)):
    clf = target_clf_list[clf_idx].clf

    fig.subplot(2, int(ceil(len(target_clf_list) / 2)), clf_idx + 1)
    fig.sp.title(target_clf_list[clf_idx].clf_name)

    fig.sp.plot_decision_regions(clf, n_grid_points=200)
    fig.sp.grid(grid_on=False)

    s_idx = ts.Y.find(ts.Y != y_target)

    for pt in s_idx[:10]:  # Plot the translation of multiple adversarial samples
        pt_segment = CArray.append(ts.X[pt, :], adv_ds.X[pt, :], axis=0)
        fig.sp.plot_path(pt_segment)

    acc = metric.performance_score(
        y_true=ts[s_idx[:10], :].Y, y_pred=clf.predict(adv_ds[s_idx[:10], :].X))

    fig.sp.text(0.01, 0.01, "Transfer attack success: {:.1%}".format(acc),
                bbox=dict(facecolor='white'))

fig.show()
```



These 2D plot clearly visualize the vulnerability of the target classifiers. The adversarial examples (green stars) which are inside the `y_target = 2` green decision region, are successfully transfered.

Poisoning Attacks against Machine Learning models

In this tutorial we will experiment with **adversarial poisoning attacks** against a Support Vector Machine (SVM) with Radial Basis Function (RBF) kernel.

Poisoning attacks are performed at *train time* by injecting *carefully crafted samples* that alter the classifier decision function so that its accuracy decreases.

As in the previous tutorials, we will first create and train the classifier, evaluating its performance in the standard scenario, *i.e. not under attack*. The poisoning attack will also need a *validation set* to verify the classifier performance during the attack, so we split the training set furtherly in two.

[1]:
```
random_state = 999

n_features = 2  # Number of features
n_samples = 300  # Number of samples
centers = [[-1, -1], [+1, +1]]  # Centers of the clusters
cluster_std = 0.9  # Standard deviation of the clusters

from secml.data.loader import CDLRandomBlobs
dataset = CDLRandomBlobs(n_features=n_features,
                         centers=centers,
                         cluster_std=cluster_std,
                         n_samples=n_samples,
                         random_state=random_state).load()
```

(continues on next page)

(continued from previous page)

```
n_tr = 100  # Number of training set samples
n_val = 100  # Number of validation set samples
n_ts = 100  # Number of test set samples

# Split in training, validation and test
from secml.data.splitter import CTrainTestSplit
splitter = CTrainTestSplit(
    train_size=n_tr + n_val, test_size=n_ts, random_state=random_state)
tr_val, ts = splitter.split(dataset)
splitter = CTrainTestSplit(
    train_size=n_tr, test_size=n_val, random_state=random_state)
tr, val = splitter.split(dataset)

# Normalize the data
from secml.ml.features import CNormalizerMinMax
nmz = CNormalizerMinMax()
tr.X = nmz.fit_transform(tr.X)
val.X = nmz.transform(val.X)
ts.X = nmz.transform(ts.X)

# Metric to use for training and performance evaluation
from secml.ml.peval.metrics import CMetricAccuracy
metric = CMetricAccuracy()

# Creation of the multiclass classifier
from secml.ml.classifiers import CClassifierSVM
from secml.ml.kernels import CKernelRBF
clf = CClassifierSVM(kernel=CKernelRBF(gamma=10), C=1)

# We can now fit the classifier
clf.fit(tr.X, tr.Y)
print("Training of classifier complete!")

# Compute predictions on a test set
y_pred = clf.predict(ts.X)
```

```
Training of classifier complete!
```

Generation of Poisoning Samples

We are going to generate an adversarial example against the SVM classifier using the **gradient-based** algorithm for generating poisoning attacks proposed in:

[biggio12-icml] Biggio, B., Nelson, B. and Laskov, P., 2012. Poisoning attacks against support vector machines. In ICML 2012.

[biggio15-icml] Xiao, H., Biggio, B., Brown, G., Fumera, G., Eckert, C. and Roli, F., 2015. Is feature selection secure against training data poisoning?. In ICML 2015.

[demontis19-usenix] Demontis, A., Melis, M., Pintor, M., Jagielski, M., Biggio, B., Oprea, A., Nita-Rotaru, C. and Roli, F., 2019. Why Do Adversarial Attacks Transfer? Explaining Transferability of Evasion and Poisoning Attacks. In 28th Usenix Security Symposium, Santa Clara, California, USA.

To compute a poisoning point, a bi-level optimization problem has to be solved, namely:

$$\max_{x_c} A(D_{val}, \mathbf{w}^*) = \sum_{j=1}^{m} \ell(y_j, \mathbf{x}_j, \mathbf{w}^*)$$
$$s.t. \mathbf{w}^* \in \underset{\mathbf{w}}{\operatorname{argmin}} L(D_{tr} \cup (\mathbf{x}_c, y_c), \mathbf{w})$$

Where $\mathbf{x_c}$ is the poisoning point, A is the attacker objective function, L is the classifier training function. Moreover, D_{tr} is the validation dataset and D_{val} is the training dataset. The former problem, along with the poisoning point $\mathbf{x}_c$ is used to train the classifier on the poisoned data, while the latter is used to evaluate the performance on the untainted data.

The former equation depends on the classifier weights, which in turns, depends on the poisoning point.

This attack is implemented in SecML by different subclasses of the `CAttackPoisoning`. For the purpose of attacking a SVM classifier we use the `CAttackPoisoningSVM` class.

As done for the *evasion attacks*, let's specify the parameters first. We set the bounds of the attack space to the known feature space given by validation dataset. Lastly, we chose the solver parameters for this specific optimization problem.

Let's start visualizing the objective function considering a single poisoning point.

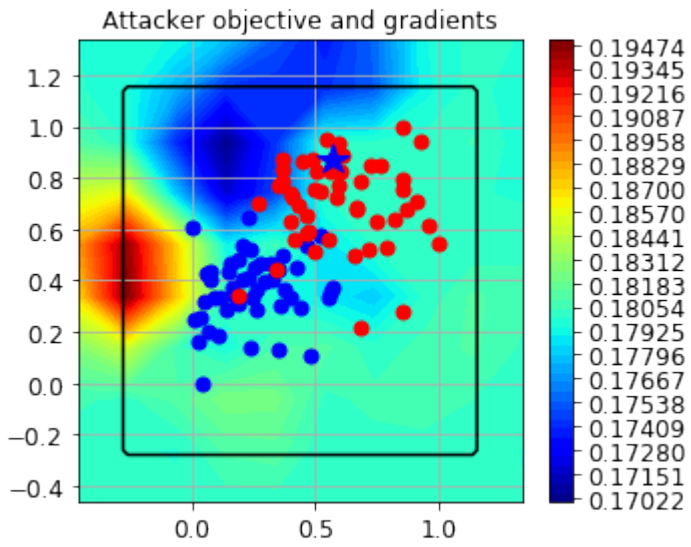
[2]:
```
lb, ub = val.X.min(), val.X.max()  # Bounds of the attack space. Can be set to `None`␣
↪for unbounded

# Should be chosen depending on the optimization problem
solver_params = {
    'eta': 0.05,
    'eta_min': 0.05,
    'eta_max': None,
    'max_iter': 100,
    'eps': 1e-6
}

from secml.adv.attacks import CAttackPoisoningSVM
pois_attack = CAttackPoisoningSVM(classifier=clf,
                                  training_data=tr,
                                  val=val,
                                  lb=lb, ub=ub,
                                  solver_params=solver_params,
                                  random_seed=random_state)

# chose and set the initial poisoning sample features and label
xc = tr[0,:].X
yc = tr[0,:].Y
pois_attack.x0 = xc
pois_attack.xc = xc
pois_attack.yc = yc

print("Initial poisoning sample features: {:}".format(xc.ravel()))
print("Initial poisoning sample label: {:}".format(yc.item()))

from secml.figure import CFigure
# Only required for visualization in notebooks
%matplotlib inline

fig = CFigure(4,5)

grid_limits = [(lb - 0.1, ub + 0.1),
```

(continues on next page)

(continued from previous page)

```
                    (lb - 0.1, ub + 0.1)]

fig.sp.plot_ds(tr)

# highlight the initial poisoning sample showing it as a star
fig.sp.plot_ds(tr[0,:], markers='*', markersize=16)

fig.sp.title('Attacker objective and gradients')
fig.sp.plot_fun(
    func=pois_attack.objective_function,
    grid_limits=grid_limits, plot_levels=False,
    n_grid_points=10, colorbar=True)

# plot the box constraint
from secml.optim.constraints import CConstraintBox
box = fbox = CConstraintBox(lb=lb, ub=ub)
fig.sp.plot_constraint(box, grid_limits=grid_limits,
                       n_grid_points=10)

fig.tight_layout()
fig.show()
```

```
Initial poisoning sample features: CArray([0.568353 0.874521])
Initial poisoning sample label: 1
```



Now, we set the desired number of adversarial points to generate, 20 in this example.

[3]:
```
n_poisoning_points = 20  # Number of poisoning points to generate
pois_attack.n_points = n_poisoning_points

# Run the poisoning attack
print("Attack started...")
pois_y_pred, pois_scores, pois_ds, f_opt = pois_attack.run(ts.X, ts.Y)
print("Attack complete!")

# Evaluate the accuracy of the original classifier
acc = metric.performance_score(y_true=ts.Y, y_pred=y_pred)
```

(continues on next page)

(continued from previous page)

```
# Evaluate the accuracy after the poisoning attack
pois_acc = metric.performance_score(y_true=ts.Y, y_pred=pois_y_pred)

print("Original accuracy on test set: {:.2%}".format(acc))
print("Accuracy after attack on test set: {:.2%}".format(pois_acc))
```

```
Attack started...
Attack complete!
Original accuracy on test set: 94.00%
Accuracy after attack on test set: 88.00%
```

We can see that the classifiers has been successfully attacked. To increase the attack power, more poisoning points can be crafted, at the expense of a much slower optimization process.

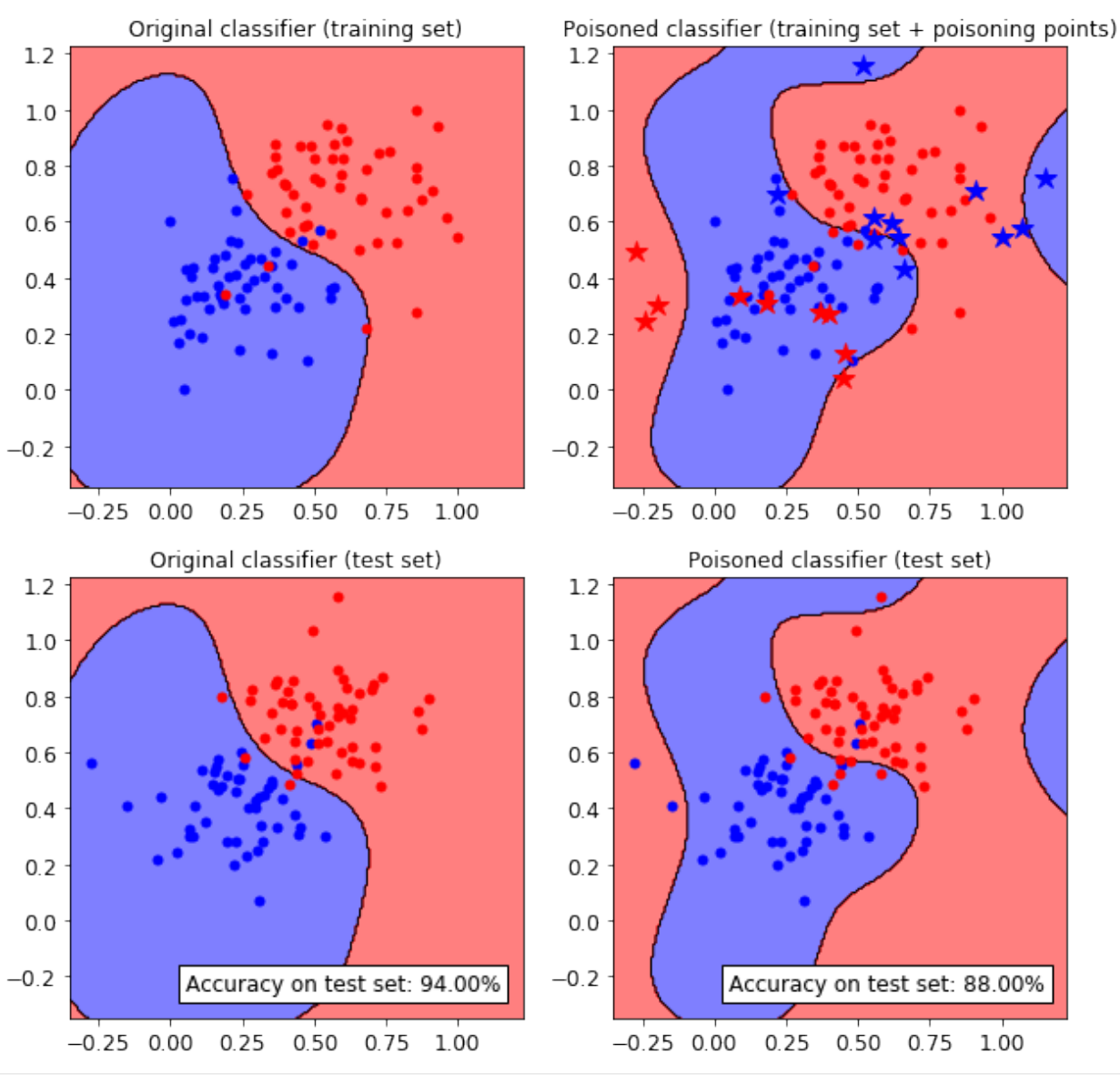
Let's now visualize the attack on a 2D plane. We need to train a copy of the original classifier on the join between the training set and the poisoning points.

[4]:
```
# Training of the poisoned classifier
pois_clf = clf.deepcopy()
pois_tr = tr.append(pois_ds)  # Join the training set with the poisoning points
pois_clf.fit(pois_tr.X, pois_tr.Y)

# Define common bounds for the subplots
min_limit = min(pois_tr.X.min(), ts.X.min())
max_limit = max(pois_tr.X.max(), ts.X.max())
grid_limits = [[min_limit, max_limit], [min_limit, max_limit]]

fig = CFigure(10, 10)

fig.subplot(2, 2, 1)
fig.sp.title("Original classifier (training set)")
fig.sp.plot_decision_regions(
    clf, n_grid_points=200, grid_limits=grid_limits)
fig.sp.plot_ds(tr, markersize=5)
fig.sp.grid(grid_on=False)

fig.subplot(2, 2, 2)
fig.sp.title("Poisoned classifier (training set + poisoning points)")
fig.sp.plot_decision_regions(
    pois_clf, n_grid_points=200, grid_limits=grid_limits)
fig.sp.plot_ds(tr, markersize=5)
fig.sp.plot_ds(pois_ds, markers=['*', '*'], markersize=12)
fig.sp.grid(grid_on=False)

fig.subplot(2, 2, 3)
fig.sp.title("Original classifier (test set)")
fig.sp.plot_decision_regions(
    clf, n_grid_points=200, grid_limits=grid_limits)
fig.sp.plot_ds(ts, markersize=5)
fig.sp.text(0.05, -0.25, "Accuracy on test set: {:.2%}".format(acc),
            bbox=dict(facecolor='white'))
fig.sp.grid(grid_on=False)

fig.subplot(2, 2, 4)
fig.sp.title("Poisoned classifier (test set)")
fig.sp.plot_decision_regions(
    pois_clf, n_grid_points=200, grid_limits=grid_limits)
```

(continues on next page)

(continued from previous page)

```
fig.sp.plot_ds(ts, markersize=5)
fig.sp.text(0.05, -0.25, "Accuracy on test set: {:.2%}".format(pois_acc),
            bbox=dict(facecolor='white'))
fig.sp.grid(grid_on=False)

fig.show()
```



We can see how the SVM classifier decision functions *changes* after injecting the adversarial poisoning points (blue and red stars).

For more details about poisoning adversarial attacks please refer to:

[biggio18-pr] Biggio, B. and Roli, F., 2018. Wild patterns: Ten years after the rise of adversarial machine learning. In Pattern Recognition.

Evasion and Poisoning Attacks on MNIST dataset

In this tutorial we show how to load the **MNIST handwritten digits dataset** and use it to train a Support Vector Machine (SVM).

Later we are going to perform Evasion and Poisoning attacks against the trained classifier, as previosuly described in *evasion* and *poisoning* tutorials.

Training of the classifier

First, we load the dataset and train the classifier. For this tutorial, we only consider 2 digits, the 5 (five) and the 9 (nine).

[1]:
```
# NBVAL_IGNORE_OUTPUT
from secml.data.loader import CDataLoaderMNIST

# MNIST dataset will be downloaded and cached if needed
loader = CDataLoaderMNIST()
```

[2]:
```
random_state = 999

n_tr = 100  # Number of training set samples
n_val = 500  # Number of validation set samples
n_ts = 500  # Number of test set samples

digits = (5, 9)

tr_val = loader.load('training', digits=digits, num_samples=n_tr + n_val)
ts = loader.load('testing', digits=digits, num_samples=n_ts)

# Split in training and validation set
tr = tr_val[:n_tr, :]
val = tr_val[n_tr:, :]

# Normalize the features in `[0, 1]`
tr.X /= 255
val.X /= 255
ts.X /= 255

from secml.ml.classifiers import CClassifierSVM
# train SVM in the dual space, on a linear kernel, as needed for poisoning
clf = CClassifierSVM(C=10, kernel='linear')

print("Training of classifier...")
clf.fit(tr.X, tr.Y)

# Compute predictions on a test set
y_pred = clf.predict(ts.X)

# Metric to use for performance evaluation
from secml.ml.peval.metrics import CMetricAccuracy
metric = CMetricAccuracy()

# Evaluate the accuracy of the classifier
acc = metric.performance_score(y_true=ts.Y, y_pred=y_pred)
```

(continues on next page)

(continued from previous page)

```
print("Accuracy on test set: {:.2%}".format(acc))
```

```
Training of classifier...
Accuracy on test set: 93.60%
```

Evasion attack with MNIST dataset

Let's define the attack parameters. Firstly, we chose to generate an *l2* perturbation within a maximum ball of radius `eps = 2.5` from the initial points. Secondly, we also add a low/upper bound as our feature space is limited in `[0, 1]`. Lastly, as we are not interested in generating adversarial examples for a specific class, we perform an error-generic attack by setting `y_target = None`.

Please note that the attack using the MNIST dataset may take a while (up to a few minutes) depending on the machine the script is run on.

[3]:
```
# For simplicity, let's attack a subset of the test set
attack_ds = ts[:25, :]

noise_type = 'l2'  # Type of perturbation 'l1' or 'l2'
dmax = 2.5  # Maximum perturbation
lb, ub = 0., 1.  # Bounds of the attack space. Can be set to `None` for unbounded
y_target = None  # None if `error-generic` or a class label for `error-specific`

# Should be chosen depending on the optimization problem
solver_params = {
    'eta': 0.5,
    'eta_min': 2.0,
    'eta_max': None,
    'max_iter': 100,
    'eps': 1e-6
}

from secml.adv.attacks import CAttackEvasionPGDLS
pgd_ls_attack = CAttackEvasionPGDLS(classifier=clf,
                                    double_init_ds=tr,
                                    distance=noise_type,
                                    dmax=dmax,
                                    solver_params=solver_params,
                                    y_target=y_target)

print("Attack started...")
eva_y_pred, _, eva_adv_ds, _ = pgd_ls_attack.run(attack_ds.X, attack_ds.Y)
print("Attack complete!")

acc = metric.performance_score(
    y_true=attack_ds.Y, y_pred=clf.predict(attack_ds.X))
acc_attack = metric.performance_score(
    y_true=attack_ds.Y, y_pred=eva_y_pred)

print("Accuracy on reduced test set before attack: {:.2%}".format(acc))
print("Accuracy on reduced test set after attack: {:.2%}".format(acc_attack))
```

```
Attack started...
Attack complete!
```

(continues on next page)

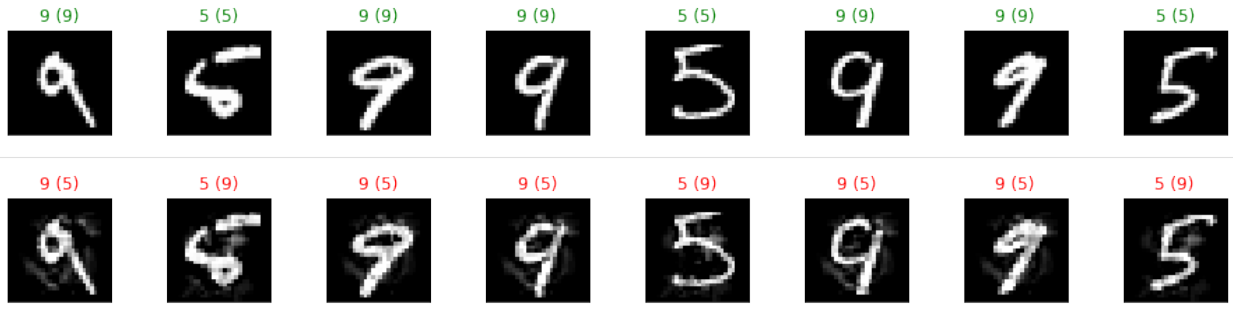
(continued from previous page)

```
Accuracy on reduced test set before attack: 100.00%
Accuracy on reduced test set after attack: 12.00%
```

We can observe how the classifier trained on the MNIST dataset has been *successfully evaded* by the adversarial examples generated by our attack.

Let's now visualize few of the adversarial examples. The first row are the original samples and the second row are the adversarial examples. Above each digit it is shown the true label and the predicted label in parenthesis.

[4]:
```
from secml.figure import CFigure
# Only required for visualization in notebooks
%matplotlib inline

# Let's define a convenience function to easily plot the MNIST dataset
def show_digits(samples, preds, labels, digs, n_display=8):
    samples = samples.atleast_2d()
    n_display = min(n_display, samples.shape[0])
    fig = CFigure(width=n_display*2, height=3)
    for idx in range(n_display):
        fig.subplot(2, n_display, idx+1)
        fig.sp.xticks([])
        fig.sp.yticks([])
        fig.sp.imshow(samples[idx, :].reshape((28, 28)), cmap='gray')
        fig.sp.title("{} ({})".format(digits[labels[idx].item()], digs[preds[idx].
↪item()]),
                     color=("green" if labels[idx].item()==preds[idx].item() else "red
↪"))
    fig.show()

show_digits(attack_ds.X, clf.predict(attack_ds.X), attack_ds.Y, digits)
show_digits(eva_adv_ds.X, clf.predict(eva_adv_ds.X), eva_adv_ds.Y, digits)
```



Poisoning attack with MNIST dataset

For poisoning attacks the parameters are much simpler. We set the the bounds of the attack space and the number of adversarial points to generate, 50 in this example. Lastly, we chose the solver parameters for this specific optimization problem.

Please note that the attack using the MNIST dataset may take a while (up to a few minutes) depending on the machine the script is run on.

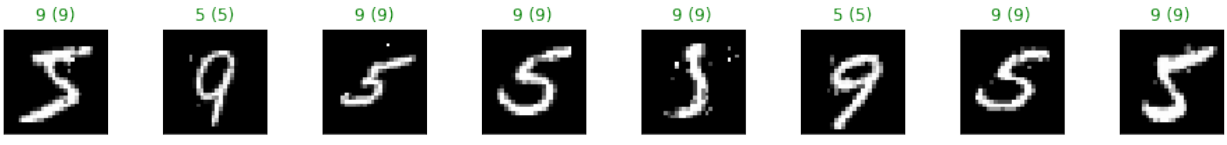
[5]:
```
lb, ub = 0., 1.  # Bounds of the attack space. Can be set to `None` for unbounded
n_poisoning_points = 15  # Number of poisoning points to generate
```

(continues on next page)

(continued from previous page)

```
# Should be chosen depending on the optimization problem
solver_params = {
    'eta': 0.25,
    'eta_min': 2.0,
    'eta_max': None,
    'max_iter': 100,
    'eps': 1e-6
}

from secml.adv.attacks import CAttackPoisoningSVM
pois_attack = CAttackPoisoningSVM(classifier=clf,
                                  training_data=tr,
                                  val=val,
                                  lb=lb, ub=ub,
                                  solver_params=solver_params,
                                  random_seed=random_state)
pois_attack.n_points = n_poisoning_points

# Run the poisoning attack
print("Attack started...")
pois_y_pred, _, pois_points_ds, _ = pois_attack.run(ts.X, ts.Y)
print("Attack complete!")

# Evaluate the accuracy of the original classifier
acc = metric.performance_score(y_true=ts.Y, y_pred=clf.predict(ts.X))
# Evaluate the accuracy after the poisoning attack
pois_acc = metric.performance_score(y_true=ts.Y, y_pred=pois_y_pred)

print("Original accuracy on test set: {:.2%}".format(acc))
print("Accuracy after attack on test set: {:.2%}".format(pois_acc))

# Training of the poisoned classifier for visualization purposes
pois_clf = clf.deepcopy()
pois_tr = tr.append(pois_points_ds)  # Join the training set with the poisoning points
pois_clf.fit(pois_tr.X, pois_tr.Y)

show_digits(pois_points_ds.X, pois_clf.predict(pois_points_ds.X),
            pois_points_ds.Y, digits)
```

```
Attack started...
Attack complete!
Original accuracy on test set: 93.60%
Accuracy after attack on test set: 50.40%
```



We can see that the classifier trained on the MNIST dataset has been successfully poisoned. To increase the attack power, more poisoning points can be crafted, at the expense of a much slower optimization process.

Let's note that the label of each adversarial example we show has been *flipped* by the attack with respect to the actual true label. Thus, the predicted label (parenthesis) by the poisoned classifier is displayed in green when *different* from the true label of the digit.

Evasion Attacks against Neural Networks on MNIST dataset

Let's continue from the *Neural Networks tutorial*, using the MNIST dataset this time.

> **Warning**
>
> Requires installation of the `pytorch` extra dependency. See *extra components* for more information.

We can use a convolutional neural network, but we need to take care of reshaping the input to the expected input size, in this case (-1, 1, 28, 28). We will see in the following how to use torchvision's `transforms` module for this purpose.

[1]:
```
import torch
from torch import nn

class MNIST3cCNN(nn.Module):
    """Model with input size (-1, 28, 28) for MNIST 3-classes dataset."""
    def __init__(self):
        super(MNIST3cCNN, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 3)

    def forward(self, x):
        x = torch.relu(torch.max_pool2d(self.conv1(x), 2))
        x = torch.relu(torch.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        x = torch.relu(self.fc1(x))
        return self.fc2(x)
```

Now we can load MNIST dataset. Remember the input shape is `(1, 1, 28, 28)`, using NCHW convention.

The input shape is an input parameter of the wrapper, since it has to take care of input reshaping before passing it to the neural network.

[2]:
```
n_ts = 1000  # number of testing set samples

from secml.data.loader import CDataLoaderMNIST
digits = (1, 5, 9)
loader = CDataLoaderMNIST()
tr = loader.load('training', digits=digits)
ts = loader.load('testing', digits=digits, num_samples=n_ts)

# Normalize the data
tr.X /= 255
ts.X /= 255
```

Now we can use again the `CClassifierPyTorch` wrapper for having the model accessible with our library. Note that we pass the input shape as input parameter for the wrapper.

[3]:
```
from torch import optim

# Random seed
torch.manual_seed(0)
```

(continues on next page)

(continued from previous page)

```
net = MNIST3cCNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(),
                      lr=0.01, momentum=0.9)

from secml.ml.classifiers import CClassifierPyTorch
clf = CClassifierPyTorch(model=net,
                         loss=criterion,
                         optimizer=optimizer,
                         epochs=20,
                         batch_size=20,
                         input_shape=(1, 28, 28),
                         random_state=0)
```

To save time, we now load from the model zoo the pre-trained model.

[4]:
```
# NBVAL_IGNORE_OUTPUT
from secml.model_zoo import load_model
clf = load_model('mnist159-cnn')
```

And now we can check how well we can classify the digits.

[5]:
```
label_torch = clf.predict(ts.X, return_decision_function=False)

from secml.ml.peval.metrics import CMetric
metric = CMetric.create('accuracy')
acc_torch = metric.performance_score(ts.Y, label_torch)

print("Model Accuracy: {}".format(acc_torch))
```

```
Model Accuracy: 0.997
```

Crafting Evasion Attacks

We can now create, as we did in notebook *MNIST tutorial*, adversarial examples against the neural network we just trained. The code is similar to the other notebook, the only difference will be the classifier that we pass to the `CAttackEvasionPGDLS` object.

[6]:
```
# For simplicity, let's attack a subset of the test set
attack_ds = ts[:10, :]

noise_type = 'l2'  # Type of perturbation 'l1' or 'l2'
dmax = 3.0  # Maximum perturbation
lb, ub = 0., 1.  # Bounds of the attack space. Can be set to `None` for unbounded
y_target = None  # None if `error-generic` or a class label for `error-specific`

# Should be chosen depending on the optimization problem
solver_params = {
    'eta': 0.5,
    'eta_min': 2.0,
    'eta_max': None,
    'max_iter': 100,
    'eps': 1e-6
}
```

(continues on next page)

(continued from previous page)

```
from secml.adv.attacks import CAttackEvasionPGDLS
pgd_ls_attack = CAttackEvasionPGDLS(classifier=clf,
                                    double_init_ds=tr,
                                    distance=noise_type,
                                    dmax=dmax,
                                    solver_params=solver_params,
                                    y_target=y_target)

print("Attack started...")
eva_y_pred, _, eva_adv_ds, _ = pgd_ls_attack.run(attack_ds.X, attack_ds.Y)
print("Attack complete!")
```

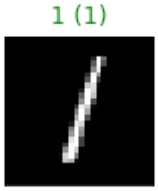
```
Attack started...
Attack complete!
```

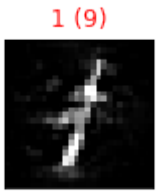
[7]:
```
acc = metric.performance_score(
    y_true=attack_ds.Y, y_pred=clf.predict(attack_ds.X))
acc_attack = metric.performance_score(
    y_true=attack_ds.Y, y_pred=eva_y_pred)

print("Accuracy on reduced test set before attack: {:.2%}".format(acc))
print("Accuracy on reduced test set after attack: {:.2%}".format(acc_attack))
```

```
Accuracy on reduced test set before attack: 100.00%
Accuracy on reduced test set after attack: 10.00%
```

Finally, we can display the adversarial digit along with its label.

[8]:
```
from secml.figure import CFigure
# Only required for visualization in notebooks
%matplotlib inline

# Let's define a convenience function to easily plot the MNIST dataset
def show_digits(samples, preds, labels, digs, n_display=8):
    samples = samples.atleast_2d()
    n_display = min(n_display, samples.shape[0])
    fig = CFigure(width=n_display*2, height=3)
    for idx in range(n_display):
        fig.subplot(2, n_display, idx+1)
        fig.sp.xticks([])
        fig.sp.yticks([])
        fig.sp.imshow(samples[idx, :].reshape((28, 28)), cmap='gray')
        fig.sp.title("{} ({})".format(digits[labels[idx].item()], digs[preds[idx].
↪item()]),
                     color=("green" if labels[idx].item()==preds[idx].item() else "red
↪"))
    fig.show()

show_digits(attack_ds.X[0, :], clf.predict(attack_ds.X[0, :]), attack_ds.Y[0, :],␣
↪digits)
show_digits(eva_adv_ds.X[0, :], clf.predict(eva_adv_ds.X[0, :]), eva_adv_ds.Y[0, :],␣
↪digits)
```





Evasion Attacks on ImageNet

Warning

Requires installation of the `pytorch` extra dependency. See *extra components* for more information.

Load the pretrained model

We can load a pretrained model from `torchvision`. We will use, for example, a ResNet18 model. We can load it in the same way it can be loaded in PyTorch, then we will pass the model object to the `secml` wrapper. Remember to pass the transformations to the `CClassifierPyTorch` object. We could have defined the transforms in the `torchvision.transforms.Compose` object, but this would now allow us to ensure that the boundary is respected in the input (not transformed) space.

[17]:
```
# NBVAL_IGNORE_OUTPUT
from torchvision import models

# Download and cache pretrained model from PyTorch model zoo
model = models.resnet18(pretrained=True)
```

[18]:
```
import torch
from torch import nn

from secml.data import CDataset
from secml.ml.classifiers import CClassifierPyTorch
from secml.ml.features import CNormalizerMeanStd

# Random seed
torch.manual_seed(0)

criterion = nn.CrossEntropyLoss()
optimizer = None  # the network is pretrained

# imagenet normalization
normalizer = CNormalizerMeanStd(mean=(0.485, 0.456, 0.406),
                                std=(0.229, 0.224, 0.225))
```

(continues on next page)

(continued from previous page)

```
# wrap the model, including the normalizer
clf = CClassifierPyTorch(model=model,
                         loss=criterion,
                         optimizer=optimizer,
                         epochs=10,
                         batch_size=1,
                         input_shape=(3, 224, 224),
                         softmax_outputs=False,
                         preprocess=normalizer,
                         random_state=0,
                         pretrained=True)
```

Load and classify an image

Now we can load an image from the web and obtain the classification output. We use the PIL and io module for reading the image, requests for getting the image, and matplotlib for visualization.

[19]:
```
from torchvision import transforms
transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
])

from PIL import Image
import requests
import io

# img_path = input("Insert image path:")
img_path = 'https://en.upali.ch/wp-content/uploads/2016/11/arikanischer-ruessel.jpg'
r = requests.get(img_path)
img = Image.open(io.BytesIO(r.content))

# apply transform from torchvision
img_t = transform(img)

# convert to CArray
from secml.array import CArray
batch_t = torch.unsqueeze(img_t, 0).view(-1)
batch_c = CArray(batch_t.numpy())

# prediction for the given image
preds = clf.predict(batch_c)
```

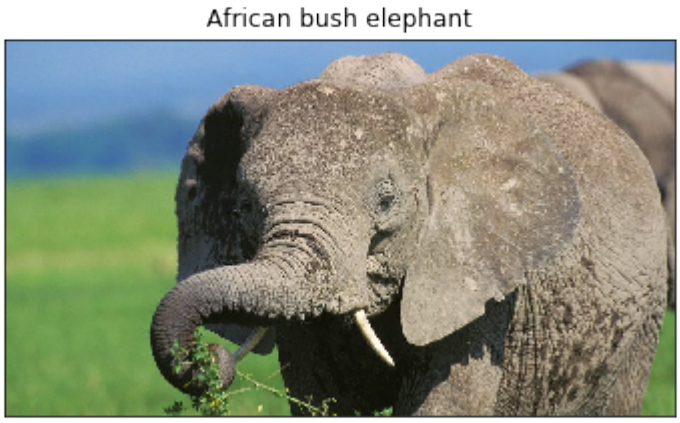
Now we have to load the ImageNet human-readable labels from a website in order to get the string label with the class name. We can display the image along with the predicted label.

[20]:
```
import json
imagenet_labels_path = "https://raw.githubusercontent.com/" \
                       "anishathalye/imagenet-simple-labels/" \
                       "master/imagenet-simple-labels.json"
r = requests.get(imagenet_labels_path)
labels = json.load(io.StringIO(r.text))
label = preds.item()
```

(continues on next page)

(continued from previous page)

```
predicted_label = labels[label]

from secml.figure import CFigure
# Only required for visualization in notebooks
%matplotlib inline

fig = CFigure()
fig.sp.imshow(img)
fig.sp.xticks([])
fig.sp.yticks([])
fig.sp.title(predicted_label)
fig.show()
```



Run the attack

We can create adversarial examples from this image, just as we did in the other notebooks. It will take no more than creating a `CAttackEvasionPGDLS` object. We should also apply the box constraint with the boundaries for the features `lb` and `ub`. Remember that this constraint will project the modified sample in the image space `[0, 1]`, ensuring the adversarial example remains in the feasible space. The constraints are applied in the input space, before the image normalization.

[21]:
```
noise_type = 'l2'  # Type of perturbation 'l1' or 'l2'
dmax = 5  # Maximum perturbation
lb, ub = 0.0, 1.0 # Bounds of the attack space. Can be set to `None` for unbounded
y_target = 1  # None if `error-generic` or a class label for `error-specific`

# Should be chosen depending on the optimization problem
solver_params = {
    'eta': 0.01,
    'eta_min': 2.0,
    'max_iter': 100,
    'eps': 1e-3
}

from secml.adv.attacks import CAttackEvasionPGDLS
pgd_ls_attack = CAttackEvasionPGDLS(classifier=clf,
                                    double_init=False,
                                    distance=noise_type,
                                    dmax=dmax,
```

(continues on next page)

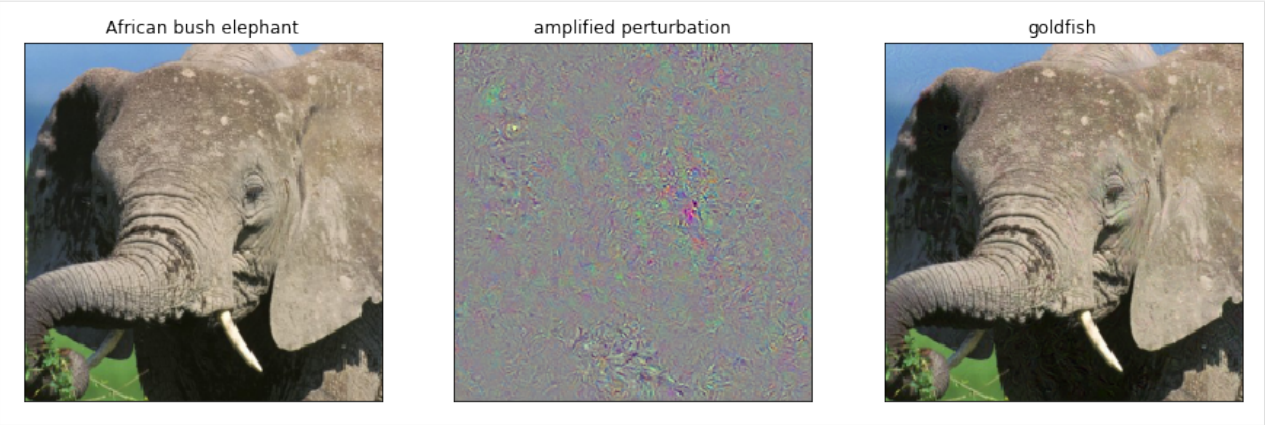
(continued from previous page)

```
                                    solver_params=solver_params,
                                    y_target=y_target,
                                    lb=lb, ub=ub)

print("Attack started...")
eva_y_pred, _, eva_adv_ds, _ = pgd_ls_attack.run(batch_c, label)
print("Attack complete!")

adv_label = labels[clf.predict(eva_adv_ds.X).item()]
```

```
Attack started...
Attack complete!
```

Now we can visualize the original (not preprocessed) image and the modified one, along with the perturbation (that will be amplified for visualization). Note that we have to convert the tensors back to images in RGB format.

[22]:
```
start_img = batch_c
eva_img = eva_adv_ds.X

# normalize perturbation for visualization
diff_img = start_img - eva_img
diff_img -= diff_img.min()
diff_img /= diff_img.max()

import numpy as np
start_img = np.transpose(start_img.tondarray().reshape((3, 224, 224)), (1, 2, 0))
diff_img = np.transpose(diff_img.tondarray().reshape((3, 224, 224)), (1, 2, 0))
eva_img = np.transpose(eva_img.tondarray().reshape((3, 224, 224)), (1, 2, 0))

fig = CFigure(width=15, height=5)
fig.subplot(1, 3, 1)
fig.sp.imshow(start_img)
fig.sp.title(predicted_label)
fig.sp.xticks([])
fig.sp.yticks([])

fig.subplot(1, 3, 2)
fig.sp.imshow(diff_img)
fig.sp.title("amplified perturbation")
fig.sp.xticks([])
fig.sp.yticks([])

fig.subplot(1, 3, 3)
fig.sp.imshow(eva_img)
fig.sp.title(adv_label)
fig.sp.xticks([])
fig.sp.yticks([])

fig.show()
```



Using cleverhans within SecML

In this tutorial we will show how to craft evasion attacks against machine learning models in SecML through the `Cleverhans` interface.

Warning

Requires installation of the `pytorch` and `cleverhans` extra dependencies. See *extra components* for more information.

Training the model

The first part is the same as the (*first notebook*). We load here a 2D dataset, so that we can easily plot the attack initial point and path.

[1]:
```
random_state = 999

n_features = 2  # Number of features
n_samples = 1100  # Number of samples
centers = [[-2, 0], [2, -2], [2, 2]]  # Centers of the clusters
cluster_std = 0.75  # Standard deviation of the clusters
n_classes = len(centers)

from secml.data.loader import CDLRandomBlobs

dataset = CDLRandomBlobs(n_features=n_features,
                         centers=centers,
                         cluster_std=cluster_std,
                         n_samples=n_samples,
                         random_state=random_state).load()

n_tr = 1000  # Number of training set samples
n_ts = 100  # Number of test set samples

# Split in training and test
from secml.data.splitter import CTrainTestSplit

splitter = CTrainTestSplit(
```

(continues on next page)

(continued from previous page)

```
    train_size=n_tr, test_size=n_ts, random_state=random_state)
tr, ts = splitter.split(dataset)

# Normalize the data
from secml.ml.features import CNormalizerMinMax

nmz = CNormalizerMinMax()
tr.X = nmz.fit_transform(tr.X)
ts.X = nmz.transform(ts.X)

# Metric to use for training and performance evaluation
from secml.ml.peval.metrics import CMetricAccuracy

metric = CMetricAccuracy()

# Creation of the multiclass classifier
import torch
from torch import nn

class Net(nn.Module):
    """
    Model with input size (-1, 5) for blobs dataset
    with 5 features
    """
    def __init__(self, n_features, n_classes):
        """Example network."""
        super(Net, self).__init__()
        self.fc1 = nn.Linear(n_features, 50)
        self.fc2 = nn.Linear(50, n_classes)

    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x

# Random seed for PyTorch
torch.manual_seed(random_state)

# torch model creation
net = Net(n_features=n_features, n_classes=n_classes)

from torch import optim
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(),
                      lr=0.001, momentum=0.9)

# wrap torch model in CClassifierPyTorch class
from secml.ml.classifiers import CClassifierPyTorch
clf = CClassifierPyTorch(model=net,
                         loss=criterion,
                         optimizer=optimizer,
                         input_shape=(n_features,),
                         epochs=5,
                         random_state=random_state)

# We can now fit the classifier
clf.fit(tr.X, tr.Y)
```

(continues on next page)

(continued from previous page)

```
# Compute predictions on a test set
y_pred = clf.predict(ts.X)

# Evaluate the accuracy of the classifier
acc = metric.performance_score(y_true=ts.Y, y_pred=y_pred)

print("Accuracy on test set: {:.2%}".format(acc))
```

```
Accuracy on test set: 100.00%
```

Preparing the attacks

Now that we have the model we can prepare the attacks. We will test several attack algorithm from cleverhans library.

We can specify a starting point for the attacks, we select a point from the class 1, which is in the lower right-corner of the 2D plane. As always, we can define a box for the attack in order to comply with the feature range (`[0, 1]`) and a maximum distance, as well as the target class (as always, specifying `y_target=None` will produce an untargeted attack).

[2]:
```
from secml.array import CArray

# x0, y0 = ts[5, :].X, ts[5, :].Y  # Initial sample
x0, y0 = CArray([0.7, 0.4]), CArray([1])
lb, ub = 0, 1
dmax = 0.4
y_target = 2
```

We can finally specify the parameters for the attacks. We can compare the paths of several attacks, for which the parameters can be found in cleverhans docs.

We are going to use the following attacks:

- FGM Goodfellow IJ, Shlens J, Szegedy C. Explaining and Harnessing Adversarial Examples. arXiv:14126572 [cs, stat] [Internet]. 2014
- PGD Kurakin A, Goodfellow I, Bengio S. Adversarial examples in the physical world. arXiv:160702533 [cs, stat] [Internet]. 2017
- MIM Dong Y, Liao F, Pang T, Su H, Zhu J, Hu X, et al. Boosting Adversarial Attacks with Momentum. In: 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition [Internet]. Salt Lake City, UT: IEEE; 2018
- CW Carlini N, Wagner D. Towards Evaluating the Robustness of Neural Networks. arXiv:160804644 [cs] [Internet]. 2016

[3]:
```
from cleverhans.attacks import CarliniWagnerL2, ProjectedGradientDescent, \
    MomentumIterativeMethod,FastGradientMethod

from collections import namedtuple
Attack = namedtuple('Attack', 'attack_cls short_name attack_params')

attacks = [
    Attack(FastGradientMethod, 'FGM', {'eps': dmax,
                                       'clip_max': ub,
                                       'clip_min': lb,
```

(continues on next page)

(continued from previous page)

```
                                        'ord': 2}),
    Attack(ProjectedGradientDescent, 'PGD', {'eps': dmax,
                                             'eps_iter': 0.05,
                                             'nb_iter': 50,
                                             'clip_max': ub,
                                             'clip_min': lb,
                                             'ord': 2,
                                             'rand_init': False}),
    Attack(MomentumIterativeMethod, 'MIM', {'eps': dmax,
                                            'eps_iter': 0.05,
                                            'nb_iter': 50,
                                            'clip_max': ub,
                                            'clip_min': lb,
                                            'ord': 2,
                                            'decay_factor': 1}),
    Attack(CarliniWagnerL2, 'CW2', {'binary_search_steps': 1,
                                    'initial_const': 0.2,
                                    'confidence': 10,
                                    'abort_early': True,
                                    'clip_min': lb,
                                    'clip_max': ub,
                                    'max_iterations': 50,
                                    'learning_rate': 0.1})]
```

Running the attacks

We can now run the attacks by passing them to the `CAttackEvasionCleverhans` class, which handles the attack optimization and provides useful output as the attack path and objective function. We can plot these information with the help of the `CFigure` module and its powerful APIs `plot_function` and `plot_path`.

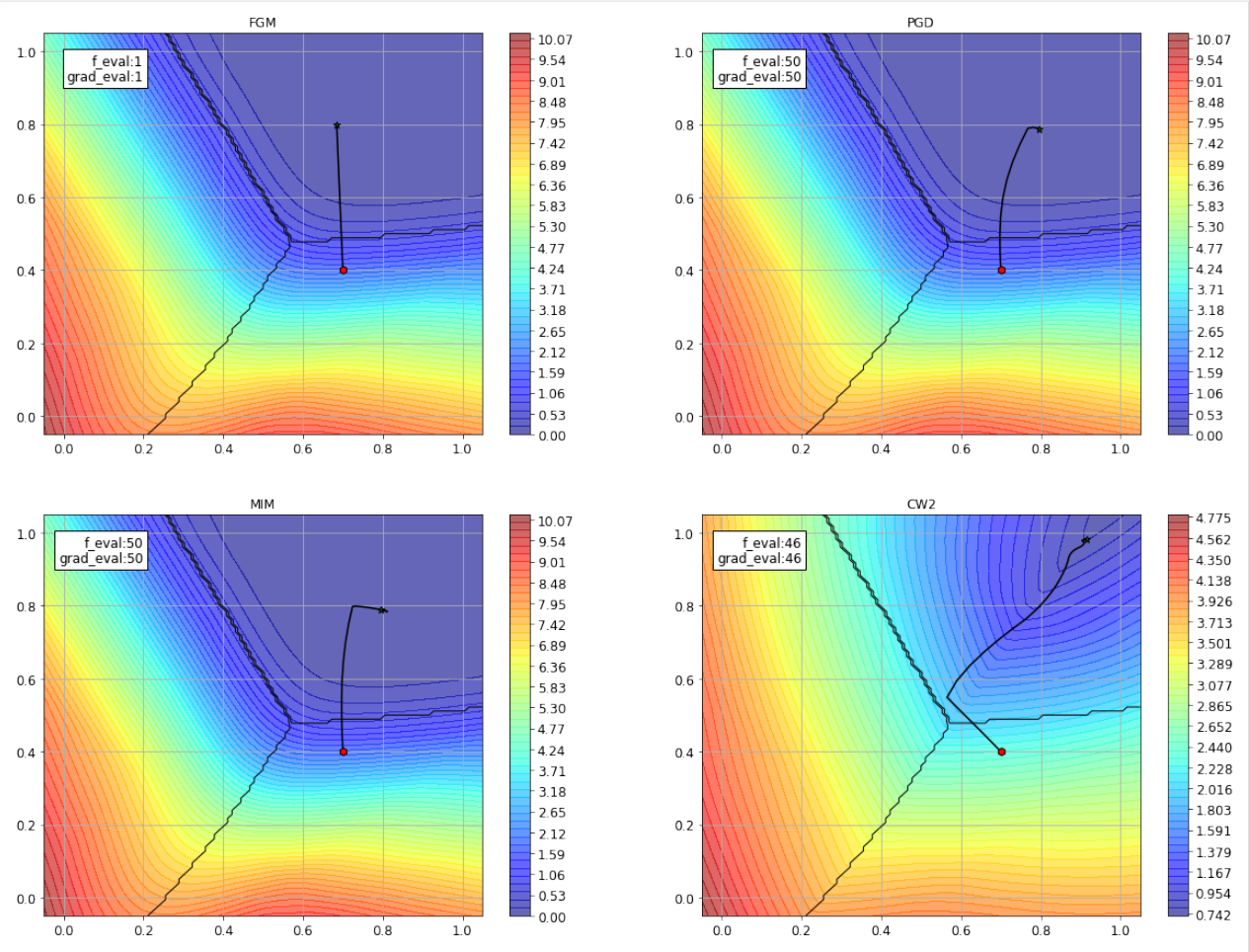
[4]:
```
from secml.figure import CFigure
# Only required for visualization in notebooks
%matplotlib inline

from secml.adv.attacks import CAttackEvasionCleverhans

fig = CFigure(width=20, height=15)

for i, attack in enumerate(attacks):
    fig.subplot(2, 2, i + 1)

    fig.sp.plot_decision_regions(clf,
                                 plot_background=False,
                                 n_grid_points=100)

    cleverhans_attack = CAttackEvasionCleverhans(
        classifier=clf,
        y_target=y_target,
        clvh_attack_class=attack.attack_cls,
        **attack.attack_params)

    # Run the evasion attack on x0
    print("Attack {:} started...".format(attack.short_name))
    y_pred_CH, _, adv_ds_CH, _ = cleverhans_attack.run(x0, y0)
    print("Attack finished!")
```

(continues on next page)

(continued from previous page)

```
    fig.sp.plot_fun(cleverhans_attack.objective_function,
                    multipoint=True, plot_levels=False,
                    n_grid_points=50, alpha=0.6)

    print("Original x0 label: ", y0.item())
    print("Adversarial example label ({:}): "
          "".format(attack.attack_cls.__name__), y_pred_CH.item())

    print("Number of classifier function evaluations: {:}"
          "".format(cleverhans_attack.f_eval))
    print("Number of classifier gradient evaluations: {:}"
          "".format(cleverhans_attack.grad_eval))

    fig.sp.plot_path(cleverhans_attack.x_seq)
    fig.sp.title(attack.short_name)
    fig.sp.text(0.2, 0.92, "f_eval:{}\ngrad_eval:{}"
                           "".format(cleverhans_attack.f_eval,
                                     cleverhans_attack.grad_eval),
                bbox=dict(facecolor='white'), horizontalalignment='right')
fig.show()
```

```
Attack FGM started...
Attack finished!
Original x0 label:  1
Adversarial example label (FastGradientMethod):  2
Number of classifier function evaluations: 1
Number of classifier gradient evaluations: 1
Attack PGD started...
Attack finished!
Original x0 label:  1
Adversarial example label (ProjectedGradientDescent):  2
Number of classifier function evaluations: 50
Number of classifier gradient evaluations: 50
Attack MIM started...
Attack finished!
Original x0 label:  1
Adversarial example label (MomentumIterativeMethod):  2
Number of classifier function evaluations: 50
Number of classifier gradient evaluations: 50
Attack CW2 started...
Attack finished!
Original x0 label:  1
Adversarial example label (CarliniWagnerL2):  2
Number of classifier function evaluations: 46
Number of classifier gradient evaluations: 46
```



Evasion Attacks on ImageNet (Advanced)

We show here how to run different evasion attacks against ResNet-18, a DNN pretrained on ImageNet. This notebook enables running also CleverHans attacks (implemented in TensorFlow) against PyTorch models.

We aim to have the image of a race car misclassified as a tiger, using the ℓ_2-norm targeted implementations of the Carlini-Wagner (CW) attack (from CleverHans), and of our PGD attack.

We also consider a variant of our PGD attack, referred to as PGD-patch, where we restrict the attacker to only change the pixels of the image corresponding to the license plate, using a box constraint (see Melis et al., *Is Deep Learning Safe for Robot Vision? Adversarial Examples against the iCub Humanoid*, ICCVW ViPAR 2017, https://arxiv.org/abs/1708.06939).

Warning

Requires installation of the `pytorch` extra dependency. See *extra components* for more information.

Load data

We start by loading the pre-trained ResNet18 model from `torchvision`, and we pass it to the `SecML` wrapper. Then, we load the ImageNet labels.

[1]:
```
# NBVAL_IGNORE_OUTPUT
from torchvision import models

# Download and cache pretrained model from PyTorch model zoo
model = models.resnet18(pretrained=True)
```

[2]:
```
import io

import numpy as np
import requests
import torch

from secml.ml.classifiers import CClassifierPyTorch
from secml.ml.features import CNormalizerMeanStd

# Set random seed for pytorch and numpy
np.random.seed(0)
torch.manual_seed(0)

# imagenet normalization
normalizer = CNormalizerMeanStd(mean=(0.485, 0.456, 0.406),
                                std=(0.229, 0.224, 0.225))

# wrap the model, including the normalizer
clf = CClassifierPyTorch(model=model,
                         input_shape=(3, 224, 224),
                         softmax_outputs=False,
                         preprocess=normalizer,
                         random_state=0,
                         pretrained=True)

# load the imagenet labels
import json

imagenet_labels_path = "https://raw.githubusercontent.com/" \
                       "anishathalye/imagenet-simple-labels/" \
                       "master/imagenet-simple-labels.json"
r = requests.get(imagenet_labels_path)
labels = json.load(io.StringIO(r.text))
```

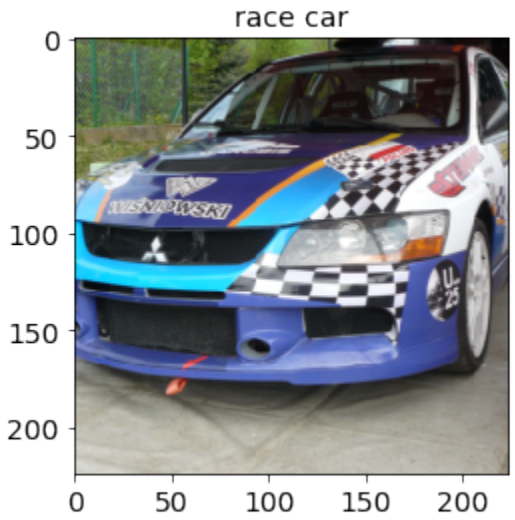
We load the image of a race car that we would like to have misclassified as a tiger and we show it, along with the classifier prediction.

[3]:
```
import numpy as np
from PIL import Image
from torchvision import transforms

from secml.array import CArray

transform = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
```

(continues on next page)

(continued from previous page)

```
    transforms.ToTensor(),
])

img_path = "https://images.pexels.com/photos/589782/pexels-photo-589782.jpeg"\
           "?cs=srgb&dl=car-mitshubishi-race-rally-589782.jpg&fm=jpg"
r = requests.get(img_path)
img = Image.open(io.BytesIO(r.content))

# apply transform from torchvision
img = transform(img)
# transform the image into a vector
img = torch.unsqueeze(img, 0).view(-1)
img = CArray(img.numpy())

# get the classifier prediction
preds = clf.predict(img)
pred_class = preds.item()
pred_label = labels[pred_class]

# create a function to show images
def plot_img(f, x, label):
    x = np.transpose(x.tondarray().reshape((3, 224, 224)), (1, 2, 0))
    f.sp.title(label)
    f.sp.imshow(x)
    return f

# show the original image
from secml.figure import CFigure
# Only required for visualization in notebooks
%matplotlib inline

fig = CFigure(height=4, width=4, fontsize=14)
plot_img(fig, img, label=pred_label)
fig.show()
```



Run the attack

We should create the attack class that we will use to compute the attack. Changing the value of the variable `attack_type`, you can choose between three different attacks: `CW`, `PGD`, and `PGD-patch`.

Note that the PGD-patch attack may take some time to run depending on the machine.

[4]:
```
attack_type = 'PGD-patch'
# attack_type = 'PGD'
# attack_type = 'CW'
```

To perform a `PGD-patch` attack that only manipulates the license plate in our car image, we need to define a proper box constraint along with its upper and lower bounds. To this end, we create the following function:

[5]:
```
def define_lb_ub(image, x_low_b, x_up_b, y_low_b, y_up_b, low_b, up_b, n_channels=3):

    # reshape the img (it is stored as a flat vector)
    image = image.tondarray().reshape((3, 224, 224))

    # assign to the lower and upper bound the same values of the image pixels
    low_b_patch = deepcopy(image)
    up_b_patch = deepcopy(image)

    # for each image channel, set the lower bound of the pixels in the
    # region defined by x_low_b, x_up_b, y_low_b, y_up_b equal to lb and
    # the upper  bound equal to up in this way the attacker will be able
    # to modify only the pixels in this region.
    for ch in range(n_channels):
         low_b_patch[ch, x_low_b:x_up_b, y_low_b:y_up_b] = low_b
         up_b_patch[ch, x_low_b:x_up_b, y_low_b:y_up_b] = up_b

    return CArray(np.ravel(low_b_patch)), CArray(np.ravel(up_b_patch))
```

We instantiate and run the attack, and show the resulting adversarial image along with its explanations, computed via integrated gradients.

[6]:
```
from copy import deepcopy

from cleverhans.attacks import CarliniWagnerL2

from secml.adv.attacks import CAttackEvasion
from secml.data import CDataset
from secml.explanation import CExplainerIntegratedGradients

lb = 0
ub = 1
target_idx = 292  # tiger

attack_id = ''
attack_params = {}

if attack_type == "CW":
    attack_id = 'e-cleverhans'
    attack_params = {'max_iterations': 50, 'learning_rate': 0.005,
                     'binary_search_steps': 1, 'confidence': 1e6,
                     'abort_early': False, 'initial_const': 0.4,
                     'n_feats': 150528, 'n_classes': 1000,
                     'y_target': target_idx,
```

(continues on next page)

(continued from previous page)

```
                    'clip_min': lb, 'clip_max': ub,
                    'clvh_attack_class': CarliniWagnerL2}

if attack_type == 'PGD':
    attack_id = 'e-pgd'
    solver_params = {
        'eta': 1e-2,
        'max_iter': 50,
        'eps': 1e-6}
    attack_params = {'double_init': False,
                     'distance': 'l2',
                     'dmax': 1.875227,
                     'lb': lb,
                     'ub': ub,
                     'y_target': target_idx,
                     'solver_params': solver_params}

if attack_type == 'PGD-patch':
    attack_id = 'e-pgd'
    # create the mask that we will use to allows the attack to modify only
    # a restricted region of the image
    x_lb = 140; x_ub = 160; y_lb = 10; y_ub = 80
    dmax_patch = 5000
    lb_patch, ub_patch = define_lb_ub(
        img, x_lb, x_ub, y_lb, y_ub, lb,ub, n_channels=3)
    solver_params = {
        'eta': 0.8,
        'max_iter': 50,
        'eps': 1e-6}

    attack_params = {'double_init': False,
                     'distance': 'l2',
                     'dmax': dmax_patch,
                     'lb': lb_patch,
                     'ub': ub_patch,
                     'y_target': target_idx,
                     'solver_params': solver_params}

attack = CAttackEvasion.create(
    attack_id,
    clf,
    **attack_params)

# run the attack
eva_y_pred, _, eva_adv_ds, _ = attack.run(img, pred_class)
adv_img = eva_adv_ds.X[0,:]

# get the classifier prediction
advx_pred = clf.predict(adv_img)
advx_label_idx = advx_pred.item()
adv_pred_label = labels[advx_label_idx]
```

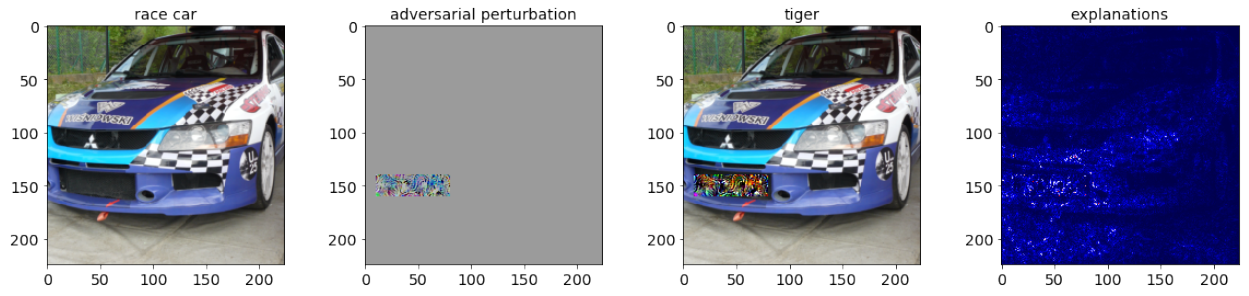
We compute the explanations for the adversarial image with respect to the target class and visualize the attack results. See the *Explainable Machine Learning* tutorial for more information.

[7]:
```
# compute the explanations w.r.t. the target class
```

(continues on next page)

(continued from previous page)

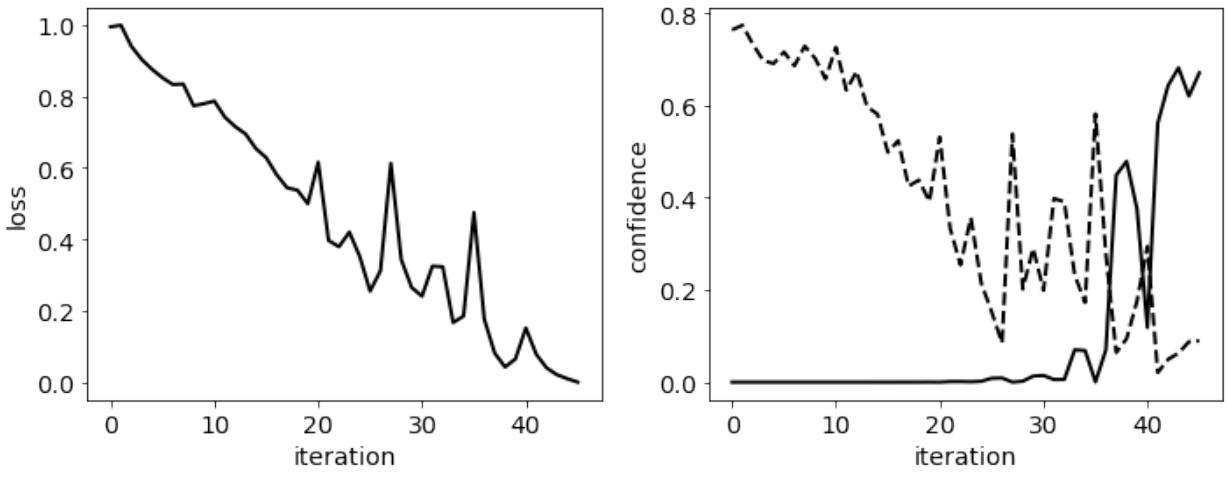
```
explainer = CExplainerIntegratedGradients(clf)
expl = explainer.explain(adv_img, y=target_idx, m=500)

fig = CFigure(height=4, width=20, fontsize=14)

fig.subplot(1, 4, 1)
# plot the original image
fig = plot_img(fig, img, label=pred_label)

# compute the adversarial perturbation
adv_noise =  adv_img - img

# normalize perturbation for visualization
diff_img = img - adv_img
diff_img -= diff_img.min()
diff_img /= diff_img.max()

# plot the adversarial perturbation
fig.subplot(1, 4, 2)
fig = plot_img(fig, diff_img, label='adversarial perturbation')

fig.subplot(1, 4, 3)
# plot the adversarial image
fig = plot_img(fig, adv_img, label=adv_pred_label)

fig.subplot(1, 4, 4)

expl = np.transpose(expl.tondarray().reshape((3, 224, 224)), (1, 2, 0))
r = np.fabs(expl[:, :, 0])
g = np.fabs(expl[:, :, 1])
b = np.fabs(expl[:, :, 2])

# Calculate the maximum error for each pixel
expl = np.maximum(np.maximum(r, g), b)
fig.sp.title('explanations')
fig.sp.imshow(expl, cmap='seismic')

fig.show()
```



Visualize and check the attack optimization

To check if the attack has properly converged to a good local minimum, we plot how the loss and the predicted confidence values of the target (solid line) and true class (dotted line) change across the attack iterations.

[8]:
```
from secml.ml.classifiers.loss import CSoftmax
from secml.ml.features.normalization import CNormalizerMinMax

n_iter = attack.x_seq.shape[0]
itrs = CArray.arange(n_iter)

# create a plot that shows the loss and the confidence during the attack iterations
# note that the loss is not available for all attacks
fig = CFigure(width=10, height=4, fontsize=14, linewidth=2)

# apply a linear scaling to have the loss in [0,1]
loss = attack.f_seq
if loss is not None:
    loss = CNormalizerMinMax().fit_transform(CArray(loss).T).ravel()
    fig.subplot(1, 2, 1)
    fig.sp.xlabel('iteration')
    fig.sp.ylabel('loss')
    fig.sp.plot(itrs, loss, c='black')

# classify all the points in the attack path
scores = clf.predict(attack.x_seq, return_decision_function=True)[1]

# we apply the softmax to the score to have value in [0,1]
scores = CSoftmax().softmax(scores)

fig.subplot(1, 2, 2)
fig.sp.xlabel('iteration')
fig.sp.ylabel('confidence')
fig.sp.plot(itrs, scores[:, pred_class], linestyle='--', c='black')
fig.sp.plot(itrs, scores[:, target_idx], c='black')

fig.tight_layout()
fig.show()
```



Deep Neural Rejection

In this tutorial we show how to train and evaluate DNR (Deep Neural Rejection), a reject-based defense against adversarial examples. For more details please refer to:

> [sotgiu20] Sotgiu, A., Demontis, A., Melis, M., Biggio, B., Fumera, G., Feng, X., Roli, F., "Deep neural rejection against adversarial examples", EURASIP J. on Info. Security 2020, 5 (2020).

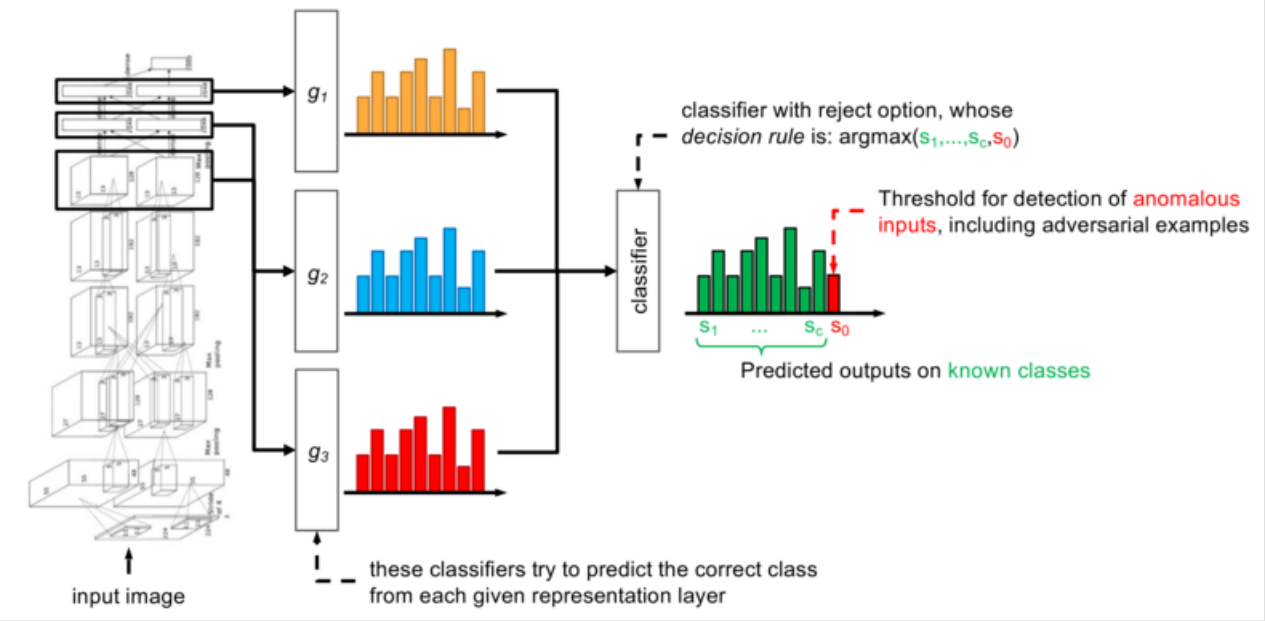
Warning

Requires installation of the `pytorch` extra dependency. See *extra components* for more information.

DNR analyzes the representations of input samples at different network layers, and rejects samples which exhibit anomalous behavior with respect to that observed from the training data at such layers.

As explained in our paper, we trained SVM classifiers with RBF kernel on each layer in order to exploit the CAP (Compact Abating Probability) property, i.e. assigned class scores decrease as samples move away from training data distributions. The outputs of these SVMs are then combined using another RBF SVM, which will provide prediction scores for each class. This classifier will reject samples if the maximum class score is not higher than the rejection threshold. We consider rejection as an additional class.

[1]:
```
from IPython.display import display, Image
display(Image(filename='_images/dnr.png'))
```



Dataset creation

In this notebook we will use the **MNIST handwritten digit dataset**.

In our experiments we follow these steps for the training phase: 1. we split the MNIST training set into two parts; 2. we train a CNN on the first half of the training set; 3. we randomly subsample 1000 samples per class from the second half of the training set, and use them to train DNR.

To save time, a CNN pre-trained on the first half of MNIST training set can be downloaded from our model zoo, and we use 5000 samples (instead of 10,000 samples) from the second half of the training set to train DNR.

[2]:
```
from secml.array import CArray
from secml.data.loader import CDataLoaderMNIST
from secml.data.selection import CPSRandom
from secml.data.splitter import CDataSplitterShuffle

# load MNIST training set and divide it in two parts
tr_data = CDataLoaderMNIST().load(ds='training')
tr_data.X /= 255.0
splitter = CDataSplitterShuffle(num_folds=1, train_size=0.5,
                                test_size=0.5, random_state=1)
splitter.compute_indices(tr_data)

# dnr training set, reduced to 5000 random samples
tr_set = tr_data[splitter.tr_idx[0], :]
tr_set = CPSRandom().select(dataset=tr_set, n_prototypes=5000, random_state=0)

# load test set
ts_set = CDataLoaderMNIST().load(ds='testing', num_samples=1000)
ts_set.X /= 255.0
```

[3]:
```
# NBVAL_IGNORE_OUTPUT
from secml.model_zoo import load_model
# load from model zoo the pre-trained net
dnn = load_model("mnist-cnn")
```

Training DNR

We now set up and fit the DNR classifier. The `CClassifierDNR` class can work with any multiclass classifier as combiner and layer classifiers. We will use RBF SVM for the aforementioned reason.

We have chosen the last three ReLu layers of the CNN. To save time, here we manually set the same SVM parameters we found with cross-validation in our paper. Note that parameters of layer classifiers can be set with `CCreator` utilities. Each layer classifier can be accessed using the name of the layer on it works.

Finally, `CClassifierDNR` provides a method to compute the reject threshold on a validation set, based on the desired rejection rate.

[4]:
```
from secml.ml.classifiers import CClassifierSVM
from secml.ml.kernels import CKernelRBF
from secml.ml.classifiers.reject import CClassifierDNR

layers = ['features:relu2', 'features:relu3', 'features:relu4']
combiner = CClassifierSVM(kernel=CKernelRBF(gamma=1), C=0.1)
layer_clf = CClassifierSVM(kernel=CKernelRBF(gamma=1e-2), C=10)
```

(continues on next page)

(continued from previous page)

```
dnr = CClassifierDNR(combiner=combiner, layer_clf=layer_clf, dnn=dnn,
                     layers=layers, threshold=-1000)
dnr.set_params({'features:relu4.C': 1, 'features:relu2.kernel.gamma': 1e-3})

print("Training started...")
dnr.fit(x=tr_set.X, y=tr_set.Y)
print("Training completed.")

# set the reject threshold in order to have 10% of rejected samples on the test set
print("Computing reject threshold...")
dnr.threshold = dnr.compute_threshold(rej_percent=0.1, ds=ts_set)
```

```
Training started...
Training completed.
Computing reject threshold...
```

Attacking DNR

We now run our new PGDExp (Projected Gradient Descent with Exponential line search) evasion attack on a sample of the test set. This attack performs an exponential line search on the distance constraint, saving gradient computations.

[5]:
```
from secml.adv.attacks import CAttackEvasionPGDExp

solver_params = {'eta': 1e-1, 'eta_min': 1e-1, 'max_iter': 30, 'eps': 1e-8}

pgd_exp = CAttackEvasionPGDExp(classifier=dnr, double_init_ds=tr_set, dmax=2,
                               distance='l2', solver_params=solver_params)

sample_idx = 10
print("Running attack...")
_ = pgd_exp.run(x=ts_set[sample_idx, :].X, y=ts_set[sample_idx, :].Y)
print("Attack completed.")
```

```
Running attack...
Attack completed.
```

Plotting attack results

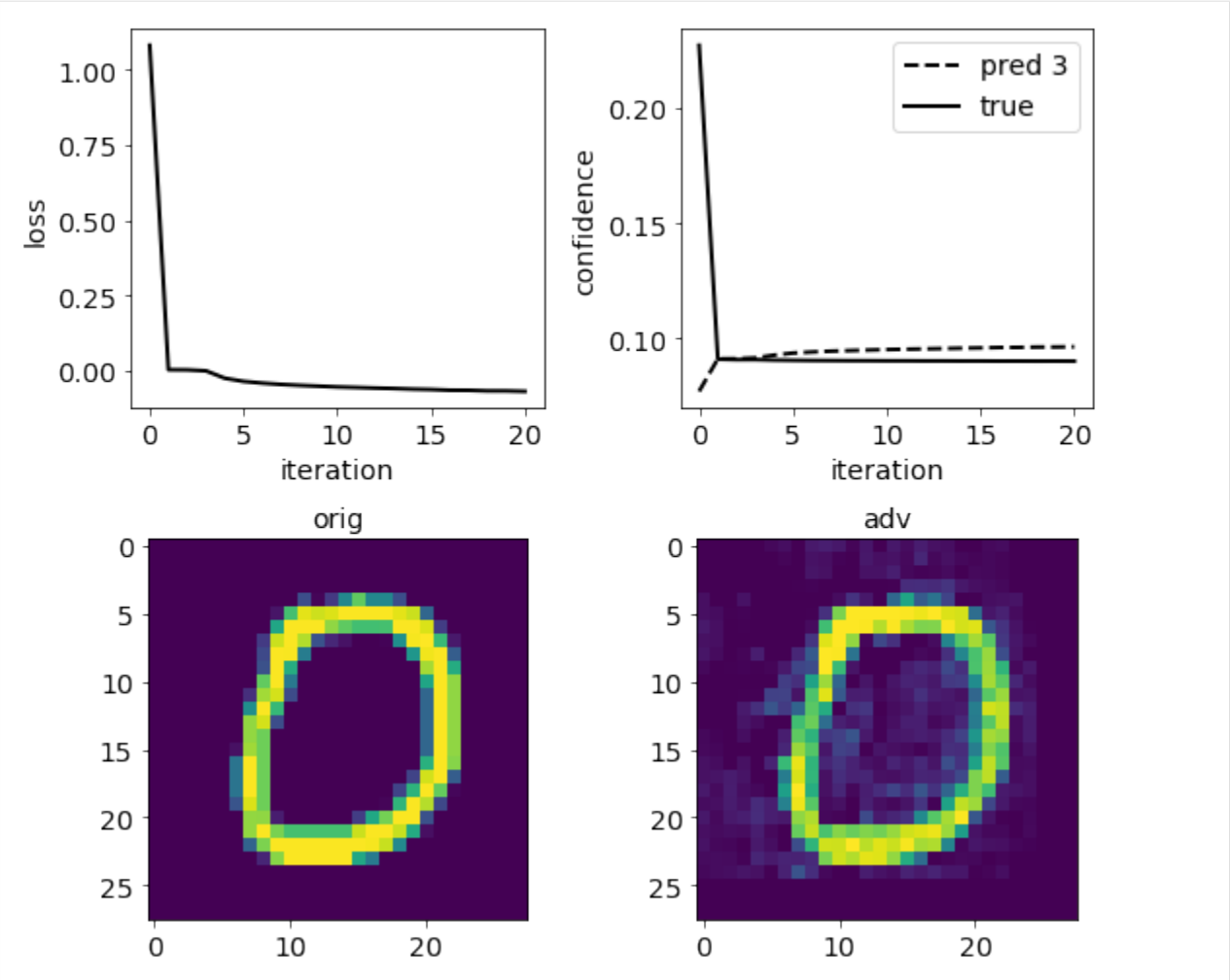
We finally plot the attack loss, the confidence of true and competing class, the original digit and the one computed by the attack. To do this, we first define an utility function, very similar to the one used in the *Evasion Attacks on ImageNet (Advanced)* tutorial.

[6]:
```
from secml.figure import CFigure
from secml.ml.classifiers.loss import CSoftmax


def plot_loss_img(attack, clf, sample_idx):
    n_iter = attack.x_seq.shape[0]
    itrs = CArray.arange(n_iter)
    true_class = ts_set.Y[sample_idx]
    fig = CFigure(width=8, height=7, fontsize=14, linewidth=2)

    fig.subplot(2, 2, 1)
    loss = attack.f_seq
```

(continues on next page)

(continued from previous page)

```
    fig.sp.xlabel('iteration')
    fig.sp.ylabel('loss')
    fig.sp.plot(itrs, loss, c='black')
    pred_classes, scores = clf.predict(attack.x_seq,
                                       return_decision_function=True)
    scores = CSoftmax().softmax(scores)

    fig.subplot(2, 2, 2)
    fig.sp.xlabel('iteration')
    fig.sp.ylabel('confidence')
    fig.sp.plot(itrs, scores[:, pred_classes[-1].item()], linestyle='--',
                c='black', label='pred {:}'.format(pred_classes[-1].item()))
    fig.sp.plot(itrs, scores[:, true_class], c='black', label='true')
    fig.sp.legend()

    fig.subplot(2, 2, 3)
    fig.sp.title('orig')
    fig.sp.imshow(ts_set.X[sample_idx, :].tondarray().reshape((28, 28)))

    fig.subplot(2, 2, 4)
    fig.sp.title('adv')
    fig.sp.imshow(attack.x_seq[-1, :].tondarray().reshape((28, 28)))

    fig.tight_layout()
    fig.show()

# Only required for visualization in notebooks
%matplotlib inline
plot_loss_img(pgd_exp, dnr, sample_idx)
```



4.7.3 Explaining Machine Learning

Interpretability of Machine Learning models has recently become a relevant research direction to more thoroughly address and mitigate the issues of adversarial examples and to better understand the potential flows of the most recent algorithm such as Deep Neural Networks.

In this tutorial, we explore different methods that SecML provides to compute *post-hoc* explanations, which consist on analyzing a trained model to understand which components such as features or training prototypes are more relevant during the decision (classification) phase.

Feature-based explanations

Feature-based explanation methods assign a value to each feature of an input sample depending on how relevant it is towards the classification decision. These relevance values are often called *attributions*.

In this tutorial, we are going to test the following *gradient-based* explanation methods:

- **Gradient**

  [baehrens2010explain] D. Baehrens, T. Schroeter, S. Harmeling, M. Kawanabe, K. Hansen, K.-R.Muller, "How to explain individual classification decisions", in: J. Mach. Learn. Res. 11 (2010) 1803-1831

- **Gradient * Input**

  [shrikumar2016not] A. Shrikumar, P. Greenside, A. Shcherbina, A. Kundaje, "Not just a blackbox: Learning important features through propagating activation differences", 2016 arXiv:1605.01713.

  [melis2018explaining] M. Melis, D. Maiorca, B. Biggio, G. Giacinto and F. Roli, "Explaining Black-box Android Malware Detection," 2018 26th European Signal Processing Conference (EUSIPCO), Rome, 2018, pp. 524-528.

- **Integrated Gradients**

  [sundararajan2017axiomatic] Sundararajan, Mukund, Ankur Taly, and Qiqi Yan. "Axiomatic Attribution for Deep Networks." Proceedings of the 34th International Conference on Machine Learning, Volume 70, JMLR. org, 2017, pp. 3319-3328.

Training of the classifier

First, we load the MSNIT dataset and we train an SVM classifier with RBF kernel.

[1]:
```
random_state = 999

n_tr = 500  # Number of training set samples
n_ts = 500  # Number of test set samples

from secml.data.loader import CDataLoaderMNIST
loader = CDataLoaderMNIST()
tr = loader.load('training', num_samples=n_tr)
ts = loader.load('testing', num_samples=n_ts)

# Normalize the features in `[0, 1]`
tr.X /= 255
ts.X /= 255

from secml.ml.classifiers.multiclass import CClassifierMulticlassOVA
from secml.ml.classifiers import CClassifierSVM
from secml.ml.kernels import CKernelRBF

clf = CClassifierMulticlassOVA( CClassifierSVM, kernel=CKernelRBF(gamma=1e-2))

print("Training of classifier...")
clf.fit(tr.X, tr.Y)

# Compute predictions on a test set
y_pred = clf.predict(ts.X)
```

(continues on next page)

(continued from previous page)

```
# Metric to use for performance evaluation
from secml.ml.peval.metrics import CMetricAccuracy
metric = CMetricAccuracy()

# Evaluate the accuracy of the classifier
acc = metric.performance_score(y_true=ts.Y, y_pred=y_pred)

print("Accuracy on test set: {:.2%}".format(acc))
```

```
Training of classifier...
Accuracy on test set: 83.80%
```

Compute the explanations

The `secml.explanation` package provides different explanation methods as subclasses of `CExplainer`. Each explainer requires as input a trained classifier.

To compute the explanation on a sample, the `.explain()` method should be used. For *gradient-based* methods, the label `y` of the class wrt the explanation should be computed is required.

The `.explain()` method will return the relevance value associated to each feature of the input sample.

[2]:
```
from secml.explanation import \
    CExplainerGradient, CExplainerGradientInput, CExplainerIntegratedGradients

explainers = (
    {
        'exp': CExplainerGradient(clf),  # Gradient
        'label': 'Gradient'
    },
    {
        'exp': CExplainerGradientInput(clf),  # Gradient * Input
        'label': 'Grad*Input'
    },
    {
        'exp': CExplainerIntegratedGradients(clf),  # Integrated Gradients
        'label': 'Int. Grads'
    },
)
```

[3]:
```
i = 123  # Test sample on which explanations should be computed
x, y = ts[i, :].X, ts[i, :].Y

print("Explanations for sample {:} (true class: {:})".format(i, y.item()))

from secml.array import CArray

for expl in explainers:

    print("Computing explanations using '{:}'...".format(
        expl['exp'].__class__.__name__))

    # Compute explanations (attributions) wrt each class
```

(continues on next page)

(continued from previous page)

```
    attr = CArray.empty(shape=(tr.num_classes, x.size))
    for c in tr.classes:

        attr_c = expl['exp'].explain(x, y=c)
        attr[c, :] = attr_c

    expl['attr'] = attr
```

```
Explanations for sample 123 (true class: 6)
Computing explanations using 'CExplainerGradient'...
Computing explanations using 'CExplainerGradientInput'...
Computing explanations using 'CExplainerIntegratedGradients'...
```

Visualize results

We now visualize the explanations computed using the different methods, in rows. In columns, we show the explanations wrt each different class.

Above the original tested sample, its true class label is shown.

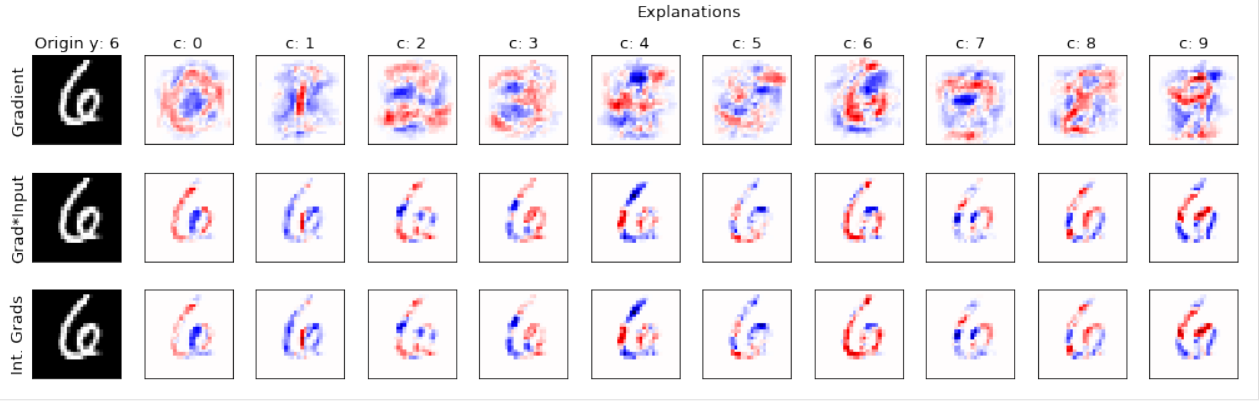
Red (blue) pixels denote positive (negative) relevance of the corresponding feature wrt the specific class.

[4]:
```
from secml.figure import CFigure
# Only required for visualization in notebooks
%matplotlib inline

fig = CFigure(height=4.5, width=14, fontsize=13)

for i, expl in enumerate(explainers):

    sp_idx = i * (tr.num_classes+1)

    # Original image
    fig.subplot(len(explainers), tr.num_classes+1, sp_idx+1)
    fig.sp.imshow(x.reshape((tr.header.img_h, tr.header.img_w)), cmap='gray')

    if i == 0:  # For the first row only
        fig.sp.title("Origin y: {:}".format(y.item()))

    fig.sp.ylabel(expl['label'])  # Label of the explainer

    fig.sp.yticks([])
    fig.sp.xticks([])

    # Threshold to plot positive and negative relevance values symmetrically
    th = max(abs(expl['attr'].min()), abs(expl['attr'].max()))

    # Plot explanations
    for c in tr.classes:

        fig.subplot(len(explainers), tr.num_classes+1, sp_idx+2+c)
        fig.sp.imshow(expl['attr'][c, :].reshape((tr.header.img_h, tr.header.img_w)),
                      cmap='seismic', vmin=-1*th, vmax=th)

        fig.sp.yticks([])
        fig.sp.xticks([])
```

(continues on next page)

(continued from previous page)

```
        if i == 0:  # For the first row only
            fig.sp.title("c: {:}".format(c))

fig.title("Explanations", x=0.55)
fig.tight_layout(rect=[0, 0.003, 1, 0.94])
fig.show()
```



For both **gradient*input** and **integrated gradients** methods we can observe a well defined area of positive (red) relevance for the explanation computed wrt the digit 6. This is expected as the true class of the tested sample is in fact 6. Moreover, a non-zero relevance value is mainly assigned to the features which are present in the tested sample, which is an expected behavior of these explanation methods.

Conversely, the **gradient** method assigns relevance to a wider area of the image, even external to the actual digit. This leads to explanations which are in many cases difficult to interpret. For this reason, more advanced explanation methods are often favored.

Prototype-based explanation

Prototype-based explanation methods select specifc samples from the training dataset to explain the behavior of machine learning models.

In this tutorial, we are going to test the explanation method proposed in:

> [koh2017understanding] Koh, Pang Wei, and Percy Liang, "Understanding black-box predictions via influence functions", in: Proceedings of the 34th International Conference on Machine Learning-Volume 70. JMLR. org, 2017.

Training of the classifier

As our implementation of the prototype-based explanation methods currently only supports binary classifiers, we load the 2-classes MNIST59 dataset and then we train a SVM classifier with RBF kernel.

[5]:
```
n_tr = 100  # Number of training set samples
n_ts = 500  # Number of test set samples

digits = (5, 9)

loader = CDataLoaderMNIST()
tr = loader.load('training', digits=digits, num_samples=n_tr)
```

(continues on next page)

(continued from previous page)

```
ts = loader.load('testing', digits=digits, num_samples=n_ts)

# Normalize the features in `[0, 1]`
tr.X /= 255
ts.X /= 255

clf = CClassifierSVM(kernel=CKernelRBF(gamma=1e-2))

print("Training of classifier...")
clf.fit(tr.X, tr.Y)

# Compute predictions on a test set
y_pred = clf.predict(ts.X)

# Metric to use for performance evaluation
metric = CMetricAccuracy()

# Evaluate the accuracy of the classifier
acc = metric.performance_score(y_true=ts.Y, y_pred=y_pred)

print("Accuracy on test set: {:.2%}".format(acc))
```

```
Training of classifier...
Accuracy on test set: 96.20%
```

Compute the influential training prototypes

The `CExplainerInfluenceFunctions` class provides the influence functions prototype-based method described previously. It requires as input the classifier to explain and its training set. It also requires the identifier of the loss used to train the classier. In the case of SVM, it is the `hinge` loss.

To compute the influence of each training sample wrt the test samples, the `.explain()` method should be used.

[6]:
```
from secml.explanation import CExplainerInfluenceFunctions

explanation = CExplainerInfluenceFunctions(
    clf, tr, outer_loss_idx='hinge')  # SVM loss is 'hinge'

print("Computing influence of each training prototype on test samples...")

infl = explanation.explain(ts.X, ts.Y)

print("Done.")
```

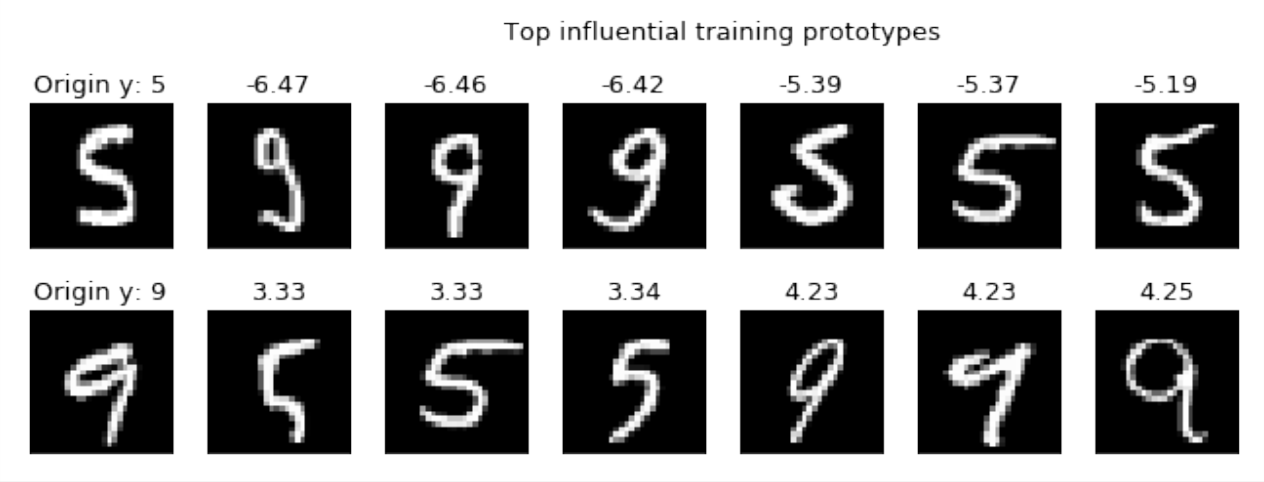
```
Computing influence of each training prototype on test samples...
Done.
```

Visualize results

We now visualize, wrt each class, the 3 most influential training prototypes for two different test samples. Above each training sample, the influence value is shown.

In addition, above the original tested samples, the true class label is shown.

[7]:
```
fig = CFigure(height=3.5, width=9, fontsize=13)

n_xc = 3  # Number of tr prototypes to plot per class

ts_list = (50, 100)  # Test samples to evaluate

infl_argsort = infl.argsort(axis=1)  # Sort influence values

for i, ts_idx in enumerate(ts_list):

    sp_idx = i * (n_xc*tr.num_classes+1)

    x, y = ts[ts_idx, :].X, ts[ts_idx, :].Y

    # Original image
    fig.subplot(len(ts_list), n_xc*tr.num_classes+1, sp_idx+1)
    fig.sp.imshow(x.reshape((tr.header.img_h, tr.header.img_w)), cmap='gray')

    fig.sp.title("Origin y: {:}".format(ts.header.y_original[y.item()]))

    fig.sp.yticks([])
    fig.sp.xticks([])

    tr_top = infl_argsort[ts_idx, :n_xc]
    tr_top = tr_top.append(infl_argsort[ts_idx, -n_xc:])

    # Plot top influential training prototypes
    for j, tr_idx in enumerate(tr_top):
        fig.subplot(len(ts_list), n_xc*tr.num_classes+1, sp_idx+2+j)
        fig.sp.imshow(tr.X[tr_idx, :].reshape((tr.header.img_h, tr.header.img_w)),␣
↪cmap='gray')

        fig.sp.title("{:.2f}".format(infl[ts_idx, tr_idx].item()))

        fig.sp.yticks([])
        fig.sp.xticks([])

fig.title("Top influential training prototypes", x=0.57)
fig.tight_layout(rect=[0, 0.003, 1, 0.92])
fig.show()
```



For both the tested samples we can observe a direct correspondence between the most influencial training prototypes and their true class. Specifically, the samples having highest (lowest) influence values are (are not) from the same true class of the tested samples.

4.7.4 secml.core

CCreator

class secml.core.c_creator.**CCreator**

Bases: object

The magnificent global superclass.

Attributes

class_type [str] Defines class type.

__super__ [str or None] String with superclass name. Can be None to explicitly NOT support *.create()* and *.load()*.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.

Continued on next page

Table 1 – continued from previous page

set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property class_type
Defines class type.

copy(*self*)
Returns a shallow copy of current class.

As shallow copy creates a new instance of current object and then insert in the new object a reference (if possible) to each attribute of the original object.

classmethod create(*class_item=None*, **args*, ***kwargs*)
This method creates an instance of a class with given type.

The list of subclasses of calling superclass is looked for any class defining *class_item = 'value'*. If found, the class type is listed.

Also a class instance can be passed as main argument. In this case the class instance is returned as is.

Parameters

class_item [str or class instance or None, optional] Type of the class to instantiate. If a class instance of cls is passed, instead, it returns the instance directly. If this is None, an instance of the classing superclass is created.

args, kwargs [optional arguments] Any other argument for the class to create. If a class instance is passed as *class_item*, optional arguments are NOT allowed.

Returns

instance_class [any class] Instance of the class having the given type (*class_type*) or the same class instance passed as input.

deepcopy(*self*)
Returns a deep copy of current class.

As deep copy is time consuming in most cases, can sometimes be acceptable to select a subset of attributes and assign them to a new instance of the current class using *.set_params*.

classmethod get_class_from_type(*class_type*)
Return the class associated with input type.

This will NOT check for classes with duplicated class type. The first class found with matching type will be returned.

Parameters

class_type [str] Type of the class which will be looked up for.

Returns

class_obj [class] Desired class, if found. This is NOT an instance of the class.

get_params(*self*)
Returns the dictionary of class hyperparameters.

A hyperparameter is a PUBLIC or READ/WRITE attribute.

get_state(*self*)
Returns the object state dictionary.

Returns

dict Dictionary containing the state of the object.

classmethod get_subclasses()
Get all the subclasses of the calling class.

Returns

subclasses [list of tuple] The list containing a tuple (class.__name__, class) for each subclass of calling class. Keep in mind that in Python each class is a "subclass" of itself.

classmethod list_class_types()
This method lists all types of available subclasses of calling one.

The list of subclasses of calling superclass is looked for any class defining *class_item = 'value'*. If found, the class type is listed.

Returns

types [list] List of the types of available subclasses of calling class.

classmethod load(*path*)
Loads object from file.

This function loads an object from file (with pickle).

The object can be correctly loaded in the following cases:

- loaded and calling class have the same type.
- calling class is the superclass of the loaded class's package.
- calling class is *.CCreator*.

Parameters

path [str] Path of the target object file.

load_state(*self*, *path*)
Sets the object state from file.

Parameters

path [str] The full path of the file from which to load the object state.

See also:

set_state Sets the object state using input dictionary.

property logger
Logger for current object.

save(*self*, *path*)
Save class object to file.

This function stores an object to file (with pickle).

.load() can be used to restore the object later.

Parameters

path [str] Path of the target object file.

Returns

obj_path [str] The full path of the stored object.

save_state(*self*, *path*)
Store the object state to file.

Parameters

path [str] Path of the file where to store object state.

Returns

str The full path of the stored object.

See also:

get_state Returns the object state dictionary.

set(*self*, *param_name*, *param_value*, *copy=False*)
Set a parameter of the class.

Only writable attributes of the class, i.e. PUBLIC or READ/WRITE, can be set.

The following checks are performed before setting:

- if *param_name* is an attribute of current class, set directly;
- **else, iterate over __dict__ and look for a class attribute** having the desired parameter as an attribute;
- **else, if attribute is not found on the 2nd level,** raise AttributeError.

If possible, a reference to the attribute to set is assigned. Use *copy=True* to always make a deepcopy before set.

Parameters

param_name [str] Name of the parameter to set.

param_value [any] Value to set for the parameter.

copy [bool] By default (False) a reference to the parameter to assign is set. If True or a reference cannot be extracted, a deepcopy of the parameter value is done first.

set_params(*self*, *params_dict*, *copy=False*)
Set all parameters passed as a dictionary {key: value}.

This function natively takes as input the dictionary created by *.get_params*. Only parameters, i.e. PUBLIC or READ/WRITE attributes, can be set.

For more information on the setting behaviour see *.CCreator.set*.

If possible, a reference to the parameter to set is assigned. Use *copy=True* to always make a deepcopy before set.

Parameters

params_dict [dict] Dictionary of parameters to set.

copy [bool] By default (False) a reference to the parameter to assign is set. If True or a reference cannot be extracted, a deepcopy of the parameter is done first.

See also:

get_params returns the dictionary of class parameters.

set_state(*self*, *state_dict*, *copy=False*)
Sets the object state using input dictionary.

Only readable attributes of the class, i.e. PUBLIC or READ/WRITE or READ ONLY, can be set.

If possible, a reference to the attribute to set is assigned. Use *copy=True* to always make a deepcopy before set.

Parameters

state_dict [dict] Dictionary containing the state of the object.

copy [bool, optional] By default (False) a reference to the attribute to assign is set. If True or a reference cannot be extracted, a deepcopy of the attribute is done first.

static timed(*msg=None*)
Timer decorator.

Returns a decorator that can be used to measure execution time of any method. Performance data will be stored inside the class logger. Messages will be logged using the INFO logging level. As this decorator accepts optional arguments, must be called as a method. See examples.

Parameters

msg [str or None, optional] Custom message to display when entering the timed block. If None, "Entering timed block *method_name*..." will printed.

property verbose
Verbosity level of logger output.

Available levels are: 0 = no verbose output 1 = info-level logging 2 = debug-level logging

secml.core.c_creator.**has_super**(*cls*)
Returns True if input class *__super__* is not None.

__super__ is defined and not None for class trees having a main superclass and one or more inherited classes.

Parameters

cls [obj] Any class or class isntance.

secml.core.c_creator.**import_class_types**(*classes*)
Returns types associated with input list of classes.

Abstract properties are ignored.

Returns

types [list] List of class types associated with input list of classes.

secml.core.c_creator.**import_package_classes**(*cls*)
Get all the classes inside a package.

Returns

members [list] Return all members of an object as (name, value) pairs sorted by name.

attr_utils

secml.core.attr_utils.**as_public**(*attr*)
Return the public name associated with a protected attribute.

Examples

```
>>> from secml.core.attr_utils import as_public
```

```
>>> as_public('_attr1')
'attr1'
>>> as_public('attr1')  # Public attributes are returned as is
'attr1'
>>> as_public('__attr1')  # This is NOT a private attribute!
'_attr1'
```

secml.core.attr_utils.**as_protected**(*attr*)
Return the protected name associated with a public attribute.

Examples

```
>>> from secml.core.attr_utils import as_protected
```

```
>>> as_protected('attr1')
'_attr1'
>>> as_protected('__attr1')
'_attr1'
>>> as_protected('_attr1')  # Protected attributes are returned as is
'_attr1'
```

secml.core.attr_utils.**has_protected**(*obj*, *attr*)
True if attribute is a protected attribute of class.

Parameters

obj [object] Target class instance.

attr [str] Name of the attribute to check.

secml.core.attr_utils.**get_protected**(*obj_class*, *attr*, *default=<no value>*)
Return the protected attribute of class.

Parameters

obj_class [class] Target class (usually extracted using obj.__class__).

attr [str] Name of the attribute to return.

default [any, optional] Value that is returned when the named attribute is not found.

secml.core.attr_utils.**as_private**(*obj_class*, *attr*)
Return the PRIVATE name associated with input attribute.

Parameters

obj_class [class] Target class (usually extracted using obj.__class__).

attr [str] Name of the target attribute.

secml.core.attr_utils.**has_private**(*obj_class*, *attr*)
True if attribute is a private attribute of class.

Parameters

obj_class [class] Target class (usually extracted using obj.__class__).

attr [str] Name of the attribute to check.

secml.core.attr_utils.**get_private**(*obj_class*, *attr*, *default=<no value>*)
Return the private attribute of class.

Parameters

obj_class [class] Target class (usually extracted using obj.__class__).

attr [str] Name of the attribute to return.

default [any, optional] Value that is returned when the named attribute is not found.

secml.core.attr_utils.**has_property**(*obj*, *attr*)
True if attribute is a property or has an associated property.

Parameters

obj [object] Any class instance.

attr [str] Name of the attribute to check.

secml.core.attr_utils.**get_property**(*obj*, *attr*)
Return the property associated with input attribute.

If no property is associated with input attribute, raise AttributeError.

Parameters

obj [object] Any class instance.

attr [str] Name of the attribute to check.

secml.core.attr_utils.**has_getter**(*obj*, *attr*)
True if an attribute has an associated getter.

Parameters

obj [object] Any class instance.

attr [str] Name of the attribute to check.

secml.core.attr_utils.**has_setter**(*obj*, *attr*)
True if an attribute has an associated setter.

Parameters

obj [object] Any class instance.

attr [str] Name of the attribute to check.

secml.core.attr_utils.**add_readonly**(*obj*, *attr*, *value=None*)
Add a READ ONLY attribute to object.

A read only attribute is defined as a protected attribute plus a getter associated with it.

Parameters

obj [object] Any class instance.

attr [str] Name of the attribute to set.

value [any, optional] Value to assign to the attribute. If not given, None is used.

secml.core.attr_utils.**add_readwrite**(*obj*, *attr*, *value=None*)
Add a READ/WRITE attribute to object.

A read/write attribute is defined as a protected attribute plus a getter AND a setter associated with it.

Parameters

obj [object] Any class instance.

attr [str] Name of the attribute to set.

value [any, optional] Value to assign to the attribute. If not given, None is used.

secml.core.attr_utils.**is_public**(*obj*, *attr*)
Return True if input attribute is PUBLIC.

A public attribute has the name without '_' as a prefix.

Parameters

obj [object] Any class instance.

attr [str] Name of the attribute to check.

secml.core.attr_utils.**is_protected**(*obj*, *attr*)
Return True if input attribute is PROTECTED.

A protected attribute has the name starting with only '_' and no getter/setter associated with it.

Parameters

obj [object] Any class instance.

attr [str] Name of the attribute to check.

secml.core.attr_utils.**is_readonly**(*obj*, *attr*)
Return True if input attribute is READ ONLY.

A read only attribute has ONLY a getter associated with it.

Parameters

obj [object] Any class instance.

attr [str] Name of the attribute to check.

secml.core.attr_utils.**is_readwrite**(*obj*, *attr*)
Return True if input attribute is READ/WRITE.

A read/write attribute has BOTH a getter AND a setter associated with it.

Parameters

obj [object] Any class instance.

attr [str] Name of the attribute to check.

secml.core.attr_utils.**is_readable**(*obj*, *attr*)
Return True if input attribute is READABLE.

A readable attribute can be one of the following:

- public
- read/write (getter/setter associated with property)
- read only (getter associated with property)

Parameters

obj [object] Any class instance.

attr [str] Name of the attribute to check.

secml.core.attr_utils.**is_writable**(*obj*, *attr*)

Return True if input attribute is WRITABLE.

A writable attribute can be one of the following:

- public
- read/write (getter/setter associated with property)

Parameters

obj [object] Any class instance.

attr [str] Name of the attribute to check.

secml.core.attr_utils.**extract_attr**(*obj*, *mode*)

Generates a sequence of attributes from an input dictionary.

This function returns a generator with the dictionary's keys having a name compatible with specified mode.

The following modalities are available:

- 'pub' -> PUBLIC (standard attribute, no '_' in the prefix)
- 'rw' -> READ/WRITE (a getter/setter is associated with it)
- 'r' -> READ ONLY (a getter is associated with it)
- 'pro' -> PROTECTED ('_' as the prefix and no getter/setter associated)

All modes can be stacked up using '+' (see examples).

Parameters

obj [any object] Any class which attributes should be extracted.

mode [str] Extraction modality. All available modalities can be combined using a plus '+'.

Notes

Sorting of the attributes in the output generator is random.

constants

secml.core.constants.**inf = inf**

Not a number.

secml.core.constants.**nan = nan**

Machine epsilon.

This is defined as the smallest number that, when added to one, yields a result different from one.

Notes

This value can be different from machine to machine, but generally yelds approximately 1.49e-08.

Examples

```
>>> from secml.core.constants import eps
>>> print(eps)
1.4901161193847656e-08
```

secml.core.constants.**eps = 1.4901161193847656e-08**
The mathematical constant e = 2.718281..., to available precision.

Examples

```
>>> from secml.core.constants import e
>>> print(e)
2.718281828459045
```

secml.core.constants.**e = 2.718281828459045**
The mathematical constant pi = 3.141592..., to available precision.

Examples

```
>>> from secml.core.constants import pi
>>> pi
3.141592653589793
```

decorators

class secml.core.decorators.**deprecated**(*version*, *extra=''*)
Bases: object

Decorator to mark a function or class as deprecated.

Issue a warning when the function is called/the class is instantiated and adds a warning to the docstring.

The optional extra argument will be appended to the deprecation message and the docstring.

Note: to use this with the default value for extra, put in an empty of parentheses: >>> from secml.core.decorators import deprecated >>> deprecated() # doctest: +ELLIPSIS <secml.core.decorators.deprecated object at ...>

```
>>> @deprecated()
... def some_function(): pass
```

Parameters

version [str] Version since which the function or class is deprecated.

extra [str, optional] Extra text to be added to the deprecation messages.

Notes

Adapted from:

- https://github.com/scikit-learn/scikit-learn/blob/master/sklearn/utils/deprecation.py
- https://wiki.python.org/moin/PythonDecoratorLibrary

Methods

`__call__`(self, obj)	Call method.

exceptions

exception secml.core.exceptions.**NotFittedError**

Bases: `ValueError`, `AttributeError`

Exception to raise if the object is used before training.

This class inherits from both ValueError and AttributeError.

Examples

```
>>> from secml.ml.classifiers import CClassifierSVM
>>> from secml.array import CArray
>>> from secml.core.exceptions import NotFittedError
>>> try:
...     CClassifierSVM().predict(CArray([[1, 2]]))
... except NotFittedError as e:
...     print(repr(e))
...
NotFittedError('this `CClassifierSVM` is not trained. Call `.fit()` first.',)
```

type_utils

secml.core.type_utils.**is_bool**(*x*)

secml.core.type_utils.**is_int**(*x*)

secml.core.type_utils.**is_intlike**(*x*)

Return True if input is integer or list/array of 1 integer.

Examples

```
>>> from secml.core.type_utils import is_intlike
```

```
>>> print(is_intlike(0))  # Standard int
True
>>> print(is_intlike(0.1))  # Standard float
False
```

```
>>> print(is_intlike(np.array([0])))  # ndarray with one int
True
>>> print(is_intlike(np.array([0.1])))  # ndarray with one float
False
```

secml.core.type_utils.**is_float**(*x*)

secml.core.type_utils.**is_floatlike**(*x*)

Return True if input is float or list/array of 1 float.

Examples

```
>>> from secml.core.type_utils import is_floatlike
```

```
>>> print(is_floatlike(0.1))  # Standard float
True
>>> print(is_floatlike(0))  # Standard int
False
```

```
>>> print(is_floatlike(np.array([0.1])))  # ndarray with one float
True
>>> print(is_floatlike(np.array([0])))  # ndarray with one int
False
```

secml.core.type_utils.**is_scalar**(*x*)

True if input is integer or float.

secml.core.type_utils.**is_scalarlike**(*x*)

True if input is scalar (int or float) or list/array of 1 real.

secml.core.type_utils.**is_inf**(*x*)

True if input is a positive/negative infinity.

Parameters

x [scalar]

Examples

```
>>> from secml.core.type_utils import is_inf
>>> from secml.core.constants import inf, nan
```

```
>>> print(is_inf(inf))
True
>>> print(is_inf(-inf))
True
```

```
>>> print(is_inf(nan))
False
```

```
>>> print(is_inf(0.1))
False
```

```
>>> from secml.array import CArray
>>> print(is_inf(CArray([inf])))  # Use `CArray.is_inf()` instead
Traceback (most recent call last):
    ...
TypeError: input must be a scalar.
```

secml.core.type_utils.**is_posinf**(*x*)

True if input is a positive infinity.

Parameters

x [scalar]

Examples

```
>>> from secml.core.type_utils import is_posinf
>>> from secml.core.constants import inf, nan
```

```
>>> print(is_posinf(inf))
True
```

```
>>> print(is_posinf(-inf))
False
```

```
>>> from secml.array import CArray
>>> print(is_posinf(CArray([inf])))  # Use `CArray.is_posinf()` instead
Traceback (most recent call last):
    ...
TypeError: input must be a scalar.
```

secml.core.type_utils.**is_neginf**(*x*)

True if input is a negative infinity.

Parameters

x [scalar]

Examples

```
>>> from secml.core.type_utils import is_neginf
>>> from secml.core.constants import inf, nan
```

```
>>> print(is_neginf(-inf))
True
```

```
>>> print(is_neginf(inf))
False
```

```
>>> from secml.array import CArray
>>> print(is_neginf(CArray([-inf])))  # Use `CArray.is_neginf()` instead
Traceback (most recent call last):
    ...
TypeError: input must be a scalar.
```

secml.core.type_utils.**is_nan**(*x*)

True if input is Not a Number (NaN).

Parameters

x [scalar]

Notes

NumPy uses the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity.

Examples

```
>>> from secml.core.type_utils import is_nan
>>> from secml.core.constants import inf, nan
```

```
>>> print(is_nan(nan))
True
```

```
>>> print(is_nan(inf))
False
```

```
>>> print(is_nan(0.1))
False
```

```
>>> from secml.array import CArray
>>> print(is_neginf(CArray([nan])))  # Use `CArray.is_nan()` instead
Traceback (most recent call last):
    ...
TypeError: input must be a scalar.
```

secml.core.type_utils.**is_list**(*x*)

secml.core.type_utils.**is_list_of_lists**(*x*)

Return True if input is a list of lists, otherwise False.

Examples

```
>>> is_list_of_lists([[1, 2], [3]])
True
>>> is_list_of_lists([[1], 2, [3]])
False
>>> is_list_of_lists([])
False
```

secml.core.type_utils.**is_ndarray**(*x*)

secml.core.type_utils.**is_scsarray**(*x*)

Returns True if input is a scipy.sparse array.

secml.core.type_utils.**is_slice**(*x*)

secml.core.type_utils.**is_str**(*x*)

secml.core.type_utils.**is_bytes**(*x*)

secml.core.type_utils.**is_tuple**(*x*)

secml.core.type_utils.**is_set**(*x*)

secml.core.type_utils.**is_dict**(*x*)

secml.core.type_utils.**to_builtin**(*x*)

Convert input to the corresponding built-in type.

Works with the following types:

- *bool*, *np.bool_* -> *bool*
- *int*, *np.integer* -> *int*
- *float, `np.floating* -> *float*
- *str*, *np.str_*, *np.unicode_* -> *str*
- *bytes*, *np.bytes_* -> *bytes*

4.7.5 secml.array

CArray

class secml.array.c_array.**CArray**(*data*, *dtype=None*, *copy=False*, *shape=None*, *tosparse=False*)

Creates an array.

Data will be stored in dense form by default.

Parameters

data [array_like or any built-in datatype] Data to be stored. Can be any array-like structure (sparse or dense) or any built-in list, scalar or string.

dtype [str or dtype, optional] Typecode or data-type to which the array is cast. If None (default), dtype is inferred from input data.

copy [bool, optional] If False (default) a reference to input data will be stored if possibile. Otherwise, a copy of original data is made first. If data is a nested sequence (a list) or dtype is different, a copy will be made anyway.

shape [int or sequence of ints, optional] Shape of the new array, e.g., '(2, 3)' or '2'.

tosparse [bool, optional] If True, input data will be converted to sparse format. Otherwise (default), if input is not a CArray, a dense array is returned, or if CArray, its format is preserved.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([[1, 2], [3, 4]]))
CArray([[1 2]
 [3 4]])
```

```
>>> print(CArray(True))
CArray([ True])
```

```
>>> print(CArray([1,0,3,4], tosparse=True))
CArray(  (0, 0)     1
  (0, 2)    3
  (0, 3)    4)
```

```
>>> print(CArray([1,2,3], dtype=float, shape=(3,1)))  # Custom dtype and shape
CArray([[1.]
 [2.]
 [3.]])
```

Attributes

T Transposed array data.

dtype Data-type of stored data.

is_vector_like True if array is vector-like.

isdense True if data is stored in DENSE form, False otherwise.

issparse True if data is stored in SPARSE form, False otherwise.

ndim Number of array dimensions.

nnz Number of non-zero values in the array.

nnz_data Return non-zero array elements.

nnz_indices Index of non-zero array elements.

shape Shape of stored data, tuple of ints.

size Size (number of elements) of array.

Methods

abs(self)	Returns array elements without sign.
all(self[, axis, keepdims])	Test whether all array elements along a given axis evaluate to True.
any(self[, axis, keepdims])	Test whether any array elements along a given axis evaluate to True.
append(self, array[, axis])	Append values to the end of an array.
apply_along_axis(self, func, axis, *args, ...)	Apply function to 1-D slices along the given axis.
arange([start, stop, step, dtype, sparse])	Return evenly spaced values within a given interval.
argmax(self[, axis])	Indices of the maximum values along an axis.

Continued on next page

Table 3 – continued from previous page

argmin(self[, axis])	Indices of the minimum values along an axis.
argsort(self[, axis, kind])	Returns the indices that would sort an array.
astype(self, dtype)	Copy of the array, casted to a specified type.
atleast_2d(self)	View original array with at least two dimensions.
binary_search(self, value)	Returns the index of each input value inside the array.
bincount(self[, minlength])	Count the number of occurrences of each value in array of non-negative ints.
ceil(self)	Return the ceiling of the input, element-wise.
clip(self, c_min, c_max)	Clip (limit) the values in an array.
comblist(list_of_list[, dtype])	Generate a cartesian product of list of list input.
concatenate(array1, array2[, axis])	Concatenate a sequence of arrays along the given axis.
cos(self)	Trigonometric cosine, element-wise.
cumsum(self[, axis, dtype])	Return the cumulative sum of the array elements along a given axis.
deepcopy(self)	Return a deepcopy of current array.
diag(self[, k])	Extract a diagonal from array or construct a diagonal array.
dot(self, array)	Dot product of two arrays.
empty(shape[, dtype, sparse])	Return a new array of given shape and type, without filling it.
exp(self)	Calculate the exponential of all elements in the input array.
eye(n_rows[, n_cols, k, dtype, sparse])	Return a 2-D array with ones on the diagonal and zeros elsewhere.
find(self, condition)	Returns vector-like array elements indices depending on condition.
find_2d(self, condition)	Returns array elements indices depending on condition.
flatten(self)	Return a flattened copy of array.
floor(self)	Return the floor of the input, element-wise.
from_iterables(iterables_list)	Build an array by chaining elements from objects in the input list.
get_data(self)	Return stored data as a standard array type.
get_nnz(self[, axis])	Counts the number of non-zero values in the array.
has_compatible_shape(self, other)	Return True if input CArray has a compatible shape.
interp(self, x_data, y_data[, return_left, ...])	One-dimensional linear interpolation.
inv(self)	Compute the (multiplicative) inverse of a square matrix.
is_inf(self)	Test element-wise for positive or negative infinity.
is_nan(self)	Test element-wise for Not a Number (NaN).
is_neginf(self)	Test element-wise for negative infinity.
is_posinf(self)	Test element-wise for positive infinity.
item(self)	Returns the single element in the array as built-in type.
linspace(start, stop[, num, endpoint, sparse])	Return evenly spaced numbers over a specified interval.
load(datafile[, dtype, arrayformat, ...])	Load array data from plain text file.
log(self)	Calculate the natural logarithm of all elements in the input array.

Continued on next page

Table 3 – continued from previous page

log10(self)	Calculate the base 10 logarithm of all elements in the input array.
logical_and(self, array)	Element-wise logical AND of array elements.
logical_not(self)	Element-wise logical NOT of array elements.
logical_or(self, array)	Element-wise logical OR of array elements.
max(self[, axis, keepdims])	Return the maximum of an array or maximum along an axis.
maximum(self, array)	Element-wise maximum of array elements.
mean(self[, axis, dtype, keepdims])	Compute the arithmetic mean along the specified axis.
median(self[, axis, keepdims])	Compute the median along the specified axis.
meshgrid(xi[, indexing])	Return coordinate matrices from coordinate vectors.
min(self[, axis, keepdims])	Return the minimum of an array or minimum along an axis.
minimum(self, array)	Element-wise minimum of array elements.
nan_to_num(self)	Replace nan with zero and inf with finite numbers.
nanargmax(self[, axis])	Indices of the maximum values along an axis ignoring NaNs.
nanargmin(self[, axis])	Indices of the minimum values along an axis ignoring NaNs
nanmax(self[, axis, keepdims])	Return the maximum of an array or maximum along an axis ignoring NaNs.
nanmin(self[, axis, keepdims])	Return the minimum of an array or minimum along an axis ignoring NaNs.
norm(self[, order])	Entrywise vector norm.
norm_2d(self[, order, axis, keepdims])	Matrix norm or vector norm along axis.
normpdf(self[, mu, sigma])	Return normal distribution function value with mean and standard deviation given for the current array values.
ones(shape[, dtype, sparse])	Return a new array of given shape and type, filled with ones.
pinv(self[, rcond])	Compute the (Moore-Penrose) pseudo-inverse of a matrix.
pow(self, exp)	Array elements raised to powers from input exponent, element-wise.
prod(self[, axis, dtype, keepdims])	Return the product of array elements over a given axis.
rand(shape[, random_state, sparse, density])	Return random floats in the half-open interval [0.0, 1.0).
randint(low[, high, shape, random_state, sparse])	Return random integers from low (inclusive) to high (exclusive).
randn(shape[, random_state])	Return a sample (or samples) from the "standard normal" distribution.
randsample(a[, shape, replace, ...])	Generates a random sample from a given array.
randuniform([low, high, shape, ...])	Return random samples from low (inclusive) to high (exclusive).
ravel(self)	Return a flattened array.
repeat(self, repeats[, axis])	Repeat elements of an array.
repmat(self, m, n)	Repeat an array M x N times.
reshape(self, newshape)	Gives a new shape to an array without changing its data.

Continued on next page

Table 3 – continued from previous page

resize(self, newshape[, constant])	Return a new array with the specified shape.
rint(self)	Round elements of the array to the nearest integer.
round(self[, decimals])	Evenly round to the given number of decimals.
save(self, datafile[, overwrite])	Save array data into plain text file.
sha1(self)	Calculate the sha1 hexadecimal hash of array.
shuffle(self)	Modify array in-place by shuffling its contents.
sign(self)	Returns element-wise sign of the array.
sin(self)	Trigonometric sine, element-wise.
sort(self[, axis, kind, inplace])	Sort an array.
sqrt(self)	Compute the positive square-root of an array, element-wise.
std(self[, axis, ddof, keepdims])	Compute the standard deviation along the specified axis.
sum(self[, axis, keepdims])	Sum of array elements over a given axis.
tocoo(self)	Return a sparse scipy.sparse.coo_matrix representation of array.
tocsc(self)	Return a sparse scipy.sparse.csc_matrix representation of array.
tocsr(self)	Return a sparse scipy.sparse.csr_matrix representation of array.
todense(self[, dtype, shape])	Converts array to dense format.
todia(self)	Return a sparse scipy.sparse.dia_matrix representation of array.
todok(self)	Return a sparse scipy.sparse.dok_matrix representation of array.
tolil(self)	Return a sparse scipy.sparse.lil_matrix representation of array.
tolist(self)	Return the array as a (possibly nested) list.
tondarray(self)	Return a dense numpy.ndarray representation of array.
tosparse(self[, dtype, shape])	Converts array to sparse format.
transpose(self)	Returns current array with axes transposed.
unique(self[, return_index, return_inverse, ...])	Find the unique elements of an array.
zeros(shape[, dtype, sparse])	Return a new array of given shape and type, filled with zeros.

property T

Transposed array data.

See also:

transpose bound method to transpose an array.

a

all(*self*, *axis=None*, *keepdims=True*)

Test whether all array elements along a given axis evaluate to True.

Axis selection is available for DENSE format only. For sparse format, logical operation is performed over all the dimensions of the array

Parameters

axis [int or None, optional, dense only] Axis or axes along which logical AND between

elements is performed. The default (axis = None) is to perform a logical AND over all the dimensions of the input array. If axis is negative, it counts from the last to the first axis.

keepdims [bool, optional, dense only] If this is set to True (default), the result will broadcast correctly against the original array. Otherwise resulting array is flattened.

Returns

bool or CArray Logical AND element-wise. If axis is None, bool is returned. Otherwise, a CArray of booleans with shape and number of dimensions consistent with the original array and the axis parameter is returned.

Notes

Not a Number (NaN), positive infinity and negative infinity evaluate to True because these are not equal to zero.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([[True,False],[True,True]], tosparse=True).all())
False
```

```
>>> print(CArray([[True,False],[True,True]]).all(axis=0))
CArray([[ True False]])
```

```
>>> print(CArray([-1,0,2,0]).all(axis=0))
CArray([ True False  True False])
>>> print(CArray([-1,0,2,0]).all(axis=1))
CArray([False])
```

```
>>> from secml.core.constants import nan, inf
>>> print(CArray([nan, inf, -inf]).all())
True
```

any (*self*, *axis=None*, *keepdims=True*)

Test whether any array elements along a given axis evaluate to True.

Axis selection is available for DENSE format only. For sparse format, logical operation is performed over all the dimensions of the array

Parameters

axis [int or None, optional, dense only] Axis or axes along which logical AND between elements is performed. The default (axis = None) is to perform a logical OR over all the dimensions of the input array. If axis is negative, it counts from the last to the first axis.

keepdims [bool, optional, dense only] If this is set to True (default), the result will broadcast correctly against the original array. Otherwise resulting array is flattened.

Returns

bool or CArray Logical OR element-wise. If axis is None, bool is returned. Otherwise, a CArray of booleans with shape and number of dimensions consistent with the original array and the axis parameter is returned.

Notes

Not a Number (NaN), positive infinity and negative infinity evaluate to True because these are not equal to zero.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([[True,False],[True,True]], tosparse=True).any())
True
```

```
>>> print(CArray([[True,False],[True,False]]).any(axis=0))
CArray([[ True False]])
```

```
>>> print(CArray([-1,0,2,0]).any(axis=0))
CArray([ True False  True False])
>>> print(CArray([-1,0,2,0]).any(axis=1))
CArray([ True])
```

```
>>> from secml.core.constants import nan, inf
>>> print(CArray([nan, inf, -inf]).any())
True
```

append(*self*, *array*, *axis=None*)

Append values to the end of an array.

Parameters

array [CArray or array_like] Second array.

axis [int or None, optional] The axis along which values are appended. If axis is None, both arrays are flattened before use.

Returns

CArray A copy of array with values appended to axis. Note that append does not occur in-place: a new array is allocated and filled. If axis is None, out is a flattened array. Always return an array with the same format of the first array.

Notes

Differently from numpy, we manage flat vectors as 2-Dimensional of shape (1, array.size). Consequently, result of appending a flat array to a flat array is 1-Dimensional only if axis=1. Appending a flat array to a 2-Dimensional array, or vice versa, always results in a 2-Dimensional array.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([[1,2],[3,4]]).append([[11],[22]]))
CArray([ 1  2  3  4 11 22])
>>> print(CArray([[1,2],[3,4]]).append([[11,22]], axis=0))
CArray([[ 1  2]
 [ 3  4]
 [11 22]])
```

```
>>> print(CArray([[1,2],[3,4]]).append(CArray([[11],[22]], tosparse=True)))
CArray([ 1  2  3  4 11 22])
>>> array = CArray([[1,2],[3,4]], tosparse=True).append([[11],[22]])
>>> print(array)
CArray(  (0, 0) 1
  (0, 1)        2
  (0, 2)        3
  (0, 3)        4
  (0, 4)        11
  (0, 5)        22)
```

```
>>> print(CArray([1,2]).append([11,22]))
CArray([ 1  2 11 22])
```

```
>>> print(CArray([1,2]).append([11,22], axis=0))
CArray([[ 1  2]
 [11 22]])
>>> print(CArray([1,2]).append([11,22], axis=1))
CArray([ 1  2 11 22])
```

apply_along_axis(*self*, *func*, *axis*, **args*, ***kwargs*)

Apply function to 1-D slices along the given axis.

func should accept 1-D arrays and return a single scalar or a 1-D array.

Only 1-D and 2-D arrays are currently supported.

Parameters

func [function] Function object to apply along the given axis. Must return a single scalar or a 1-D array.

axis [int] Axis along which to apply the function.

***args, **kwargs** [optional] Any other input value for *func*.

Returns

CArray 1-Dimensional array of size *data.shape[0]* with the output of *func* for each row in data. Datatype of output array is always float.

Examples

```
>>> from secml.array import CArray
```

```
>>> a = CArray([[1,2],[10,20],[100,200]])
```

```
>>> def return_sum(x):
...     return x.sum()
```

```
>>> print(a.apply_along_axis(return_sum, axis=0))  # Column-wise
CArray([111. 222.])
```

```
>>> print(a.apply_along_axis(return_sum, axis=1))  # Row-wise
CArray([  3.  30. 300.])
```

classmethod arange(*start=None*, *stop=None*, *step=1*, *dtype=None*, *sparse=False*)

Return evenly spaced values within a given interval.

Values are generated within the half-open interval [start, stop). For integer arguments the function is equivalent to the Python built-in range function, but returns an ndarray rather than a list.

When using a non-integer step, such as 0.1, the results will often not be consistent. It is better to use linspace for these cases.

Parameters

start [scalar, optional] Start of interval. The interval includes this value. The default start value is 0.

stop [scalar] End of interval. The interval does not include this value, except in some cases where step is not an integer and floating point round-off affects the length of the output.

step [scalar, optional] Spacing between values. For any output out, this is the distance between two adjacent values, out[i+1] - out[i]. The default step size is 1. If step is specified, start must also be given.

dtype [str or dtype, optional] The type of the output array. If dtype is not given, infer the data type from the other input arguments.

sparse [bool, optional] If False (default) a dense array will be returned. Otherwise, a sparse array is created.

Returns

CArray Array of evenly spaced values. For floating point arguments, the length of the result is ceil((stop - start)/step). Because of floating point overflow, this rule may result in the last element of out being greater than stop.

Warning: When sparse is True, array is created as dense and then converted to sparse format. Consequently, the performance of this method is not comparable to other sparse array creation routines.

See also:

CArray.linspace Evenly spaced numbers with handling of endpoints.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray.arange(4))
CArray([0 1 2 3])
```

```
>>> print(CArray.arange(4.0))
CArray([0. 1. 2. 3.])
```

```
>>> print(CArray.arange(4.0, dtype=int))
CArray([0 1 2 3])
```

```
>>> print(CArray.arange(0, 4))
CArray([0 1 2 3])
```

```
>>> print(CArray.arange(0, 4, 0.8))
CArray([0.  0.8 1.6 2.4 3.2])
```

argmax (*self*, *axis=None*)

Indices of the maximum values along an axis.

Parameters

axis [int, None, optional] If None (default), array is flattened before computing index, otherwise the specified axis is used.

Returns

int or CArray Index of the maximum of the array. If axis is None, int is returned. Otherwise, a CArray with shape and number of dimensions consistent with the original array and the axis parameter is returned.

Notes

In case of multiple occurrences of the maximum values, the indices corresponding to the first occurrence are returned.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([-1, 0, 3]).argmax())
2
```

```
>>> print(CArray([[-1, 0],[4, 3]]).argmax(axis=0))  # We return the index of␣
↪maximum for each column
CArray([[1 1]])
```

```
>>> print(CArray([[-1, 0],[4, 3]]).argmax(axis=1))  # We return the index of␣
↪maximum for each row
CArray([[1]
 [0]])
```

```
>>> print(CArray([-3,0,1,2]).argmax(axis=0))
CArray([0 0 0 0])
>>> print(CArray([-3,0,1,2]).argmax(axis=1))
CArray([3])
```

argmin (*self*, *axis=None*)

Indices of the minimum values along an axis.

Parameters

axis [int, None, optional] If None (default), array is flattened before computing index, otherwise the specified axis is used.

Returns

int or CArray Index of the minimum of the array. If axis is None, int is returned. Otherwise, a CArray with shape and number of dimensions consistent with the original array and the axis parameter is returned.

Notes

In case of multiple occurrences of the minimum values, the indices corresponding to the first occurrence are returned.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([-1, 0, 3]).argmin())
0
```

```
>>> print(CArray([[-1, 0],[4, 3]]).argmin(axis=0))  # We return the index of␣
↪minimum for each column
CArray([[0 0]])
```

```
>>> print(CArray([[-1, 0],[4, 3]]).argmin(axis=1))  # We return the index of␣
↪maximum for each row
CArray([[0]
 [1]])
```

```
>>> print(CArray([-3,0,1,2]).argmin(axis=0))
CArray([0 0 0 0])
>>> print(CArray([-3,0,1,2]).argmin(axis=1))
CArray([0])
```

argsort (*self*, *axis=-1*, *kind='quicksort'*)

Returns the indices that would sort an array.

Perform an indirect sort along the given axis using the algorithm specified by the kind keyword. It returns an array of indices of the same shape as a that index data along the given axis in sorted order.

Parameters

axis [int or None, optional] Axis along which to sort. The default is -1 (the last axis). If None, the flattened array is used.

kind [{'quicksort', 'mergesort', 'heapsort'}, optional] Sorting algorithm to use. Default 'quicksort'. For sparse arrays, only 'quicksort' is available.

Returns

CArray Array of indices that sort the array along the specified axis. In other words, array[index_array] yields a sorted array.

See also:

numpy.sort Description of different sorting algorithms.

CArray.sort In-Place sorting of array.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([0,-3,5]).argsort())
CArray([1 0 2])
```

```
>>> print(CArray([[0,-3],[5,1]]).argsort(axis=1))  # Sorting of each row
CArray([[1 0]
 [1 0]])
```

```
>>> print(CArray([[0,-3],[5,1]]).argsort(axis=None))  # Sorting the flattened␣
↪array
CArray([1 0 3 2])
```

astype(*self*, *dtype*)

Copy of the array, casted to a specified type.

Parameters

dtype [str or dtype] Typecode or data-type to which the array is cast.

Returns

CArray Copy of the original array casted to new data type.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([1, 2, 3]).astype(float))
CArray([1. 2. 3.])
```

```
>>> print(CArray([1.1, 2.1, 3.1], tosparse=True).astype(int))
CArray(  (0, 0) 1
  (0, 1)        2
  (0, 2)        3)
```

atleast_2d(*self*)

View original array with at least two dimensions.

A copy is made only if needed.

Returns

out [CArray] Array with array.ndim >= 2.

Notes

Sparse arrays are always 2 dimensional so this method returns a view (if possible) of the original array without any changes.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([1,2,3]).atleast_2d())
CArray([[1 2 3]])
```

binary_search (*self*, *value*)

Returns the index of each input value inside the array.

DENSE ARRAYS ONLY

If value is not found inside the array, the index of the closest value will be returned. Array will be flattened before search.

Parameters

value [scalar or CArray] Element or array of elements to search inside the flattened array.

Returns

int or CArray Position of input value, or the closest one, inside flattened array. If *value* is an array, a CArray with the position of each *value* element is returned.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([[0,0.1],[0.4,1.0]]).binary_search(0.3))
2
```

```
>>> print(CArray([1,2,3,4]).binary_search(10))
3
```

```
>>> print(CArray([1,2,3,4]).binary_search(CArray([-10,1,2.2,10])))
CArray([0 0 1 3])
```

bincount (*self*, *minlength=0*)

Count the number of occurrences of each value in array of non-negative ints.

Only vector like arrays of integer dtype are supported.

Parameters

minlength [int, optional] A minimum number of bins for the output.

Returns

CArray The occurrence number for every different element of array. The length of output array is equal to a.max()+1 if an argument for the parameter minlenght is not provided.

Examples

```
>>> from secml.array import CArray
```

```
>>> a = CArray([1, 2, 3, 1, 6], tosparse=True)
>>> print(a.bincount())
CArray([0 2 1 1 0 0 1])
```

ceil (*self*)

Return the ceiling of the input, element-wise.

The ceil of the scalar x is the smallest integer i, such that i >= x.

Returns

out_ceil [CArray] The ceiling of each element in x, with float dtype.

See also:

round Evenly round to the given number of decimals.

floor Return the floor of the input, element-wise.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0]).ceil())
CArray([-1. -1. -0.  1.  2.  2.  2.])
```

```
>>> # Array with dtype == int is upcasted to float before ceiling
>>> print(CArray([[-2, -1], [1, 1]], tosparse=True).ceil())
CArray(  (0, 0) -2.0
  (0, 1)        -1.0
  (1, 0)        1.0
  (1, 1)        1.0)
```

clip (*self*, *c_min*, *c_max*)

Clip (limit) the values in an array.

DENSE FORMAT ONLY

Given an interval, values outside the interval are clipped to the interval edges. For example, if an interval of [0, 1] is specified, values smaller than 0 become 0, and values larger than 1 become 1.

Parameters

c_min, c_max [int] Clipping intervals.

Returns

CArray Returns a new array containing the clipped array elements. Dtype of the output array depends on the dtype of original array and on the dtype of the clipping limits.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([[1,2],[3,4]]).clip(2, 4))
CArray([[2 2]
 [3 4]])
```

```
>>> from secml.core.constants import inf
```

```
>>> # inf is a float, array will be casted accordingly
>>> print(CArray([[1,2],[3,4]]).clip(-inf, 2))
CArray([[1. 2.]
 [2. 2.]])
```

classmethod comblist(*list_of_list*, *dtype=<class 'float'>*)

Generate a cartesian product of list of list input.

Parameters

list_of_list [list of list] 1-D arrays to form the cartesian product of.

dtype [str or dtype] Datatype of output array. Default float.

Returns

CArray 2-D array of shape (M, len(arrays)) containing cartesian products between input arrays.

Examples

```
>>> print(CArray.comblist([[1, 2, 3], [4, 5], [6, 7]]))
CArray([[1. 4. 6.]
 [1. 4. 7.]
 [1. 5. 6.]
 [1. 5. 7.]
 [2. 4. 6.]
 [2. 4. 7.]
 [2. 5. 6.]
 [2. 5. 7.]
 [3. 4. 6.]
 [3. 4. 7.]
 [3. 5. 6.]
 [3. 5. 7.]])
```

```
>>> print(CArray.comblist([[1, 2], [3]], dtype=int))
CArray([[1 3]
 [2 3]])
```

classmethod concatenate(*array1*, *array2*, *axis=1*)

Concatenate a sequence of arrays along the given axis.

The arrays must have the same shape, except in the dimension corresponding to axis (the second, by default).

This function preserves input masks if available.

Parameters

array1 [CArray or array_like] First array. If array1 is not an array, a CArray will be created before concatenating.

array2 [CArray or array_like] Second array. If array2 is not an array, a CArray will be created before concatenating.

axis [int or None, optional] The axis along which the arrays will be joined. Default is 1. If None, both arrays are ravelled before concatenation.

Returns

CArray The concatenated array. If first array is sparse, return sparse.

Notes

Differently from numpy, we manage flat vectors as 2-Dimensional of shape (1, array.size). Consequently, concatenation result of 2 flat arrays is a flat array only when axis=1.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray.concatenate([[1,2],[3,4]], [[11],[22]]))
CArray([[ 1  2 11]
 [ 3  4 22]])
>>> print(CArray.concatenate([[1,2],[3,4]], [[11,22]], axis=0))
CArray([[ 1  2]
 [ 3  4]
 [11 22]])
```

```
>>> print(CArray.concatenate([[1,2],[3,4]], CArray([[11],[22]],␣
↪tosparse=True)))
CArray([[ 1  2 11]
 [ 3  4 22]])
>>> array = CArray.concatenate(CArray([[1,2],[3,4]], tosparse=True), [[11],
↪[22]])
>>> print(array)
CArray(  (0, 0) 1
  (0, 1)        2
  (0, 2)        11
  (1, 0)        3
  (1, 1)        4
  (1, 2)        22)
```

```
>>> print(CArray.concatenate([1,2], [11,22]))
CArray([ 1  2 11 22])
```

```
>>> print(CArray.concatenate([1,2], [11,22], axis=0))
CArray([[ 1  2]
 [11 22]])
>>> print(CArray.concatenate([1,2], [11,22], axis=1))
CArray([ 1  2 11 22])
```

cos (*self*)

Trigonometric cosine, element-wise.

DENSE FORMAT ONLY

The array elements are considered angles, in radians (2π rad equals 360 degrees).

Returns

CArray New array with trigonometric cosine element-wise.

Examples

```
>>> from secml.array import CArray
>>> from secml.core.constants import pi
```

```
>>> print((CArray([0,90,180,270,360,-90,-180,-270])*pi/180).cos().round())
CArray([ 1.  0. -1. -0.  1.  0. -1. -0.])
```

```
>>> print((CArray([[45,135],[225,315]])*pi/180).cos())
CArray([[ 0.707107 -0.707107]
 [-0.707107  0.707107]])
```

cumsum(*self*, *axis=None*, *dtype=None*)

Return the cumulative sum of the array elements along a given axis.

DENSE FORMAT ONLY

Parameters

axis [int or None, optional] Axis along which the cumulative sum is computed. The default (None) is to compute the cumsum over the flattened array.

dtype [dtype or None, optional] Type of the returned array and of the accumulator in which the elements are summed. If dtype is not specified, it defaults to the dtype of a, unless a has an integer dtype with a precision less than that of the default platform integer. In that case, the default platform integer is used.

Returns

CArray New array with cumulative sum of elements. If axis is None, flat array with same size of input array. If axis is not None, same shape of input array.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([-3,0,2]).cumsum())
CArray([-3 -3 -1])
```

```
>>> print(CArray([-3,0,1,2]).cumsum(axis=0))
CArray([-3  0  1  2])
>>> print(CArray([-3,0,1,2]).cumsum(axis=1))
CArray([-3 -3 -2  0])
```

```
>>> print(CArray([[-3,0],[1,2]]).cumsum(dtype=float))
CArray([-3. -3. -2.  0.])
```

```
>>> print(CArray([[-3,0],[1,2]]).cumsum(axis=1))
CArray([[-3 -3]
 [ 1  3]])
```

d

diag (*self*, *k=0*)

Extract a diagonal from array or construct a diagonal array.

Parameters

k [int, optional] Diagonal index. Default is 0. Use k > 0 for diagonals above the main diagonal, k < 0 for diagonals below the main diagonal.

Returns

CArray The extracted diagonal or constructed diagonal dense array. If array is 2-Dimensional, returns its k-th diagonal. Depending on numpy version resulting array can be read-only or a view of the original array's diagonal. To make output array writable, use *deepcopy()*. If array is vector-like, return a 2-D array with the array on the k-th diagonal.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([[1, 2, 3], [10, 20, 30]]).diag(k=1))
CArray([ 2 30])
```

```
>>> print(CArray([[2, 1]], tosparse=True).diag())
CArray(  (0, 0) 2
  (1, 1)        1)
```

```
>>> print(CArray([1, 2, 3]).diag(k=1))
CArray([[0 1 0 0]
 [0 0 2 0]
 [0 0 0 3]
 [0 0 0 0]])
```

dot (*self*, *array*)

Dot product of two arrays.

For 2-D arrays it is equivalent to matrix multiplication. If both arrays are dense flat (rows), it is equivalent to the inner product of vectors (without complex conjugation).

Format of output array is the same of the first product argument.

Parameters

array [CArray] Second argument of dot product.

Returns

scalar or CArray Result of dot product. A CArray with the same format of first argument or scalar if *out.size == 1*.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([[1,1],[2,2]]).dot(CArray([[1,1],[0,0]], tosparse=True)))
CArray([[1 1]
 [2 2]])
```

```
>>> print(CArray([10,20]).dot(CArray([[1],[0]], tosparse=True)))
10
```

OUTER PRODUCT

```
>>> print(CArray([[10],[20]]).dot(CArray([1,0], tosparse=True)))
CArray([[10  0]
 [20  0]])
```

INNER PRODUCT BETWEEN VECTORS

```
>>> print(CArray([10,20]).dot(CArray([1,0])))
10
```

Inner product between vector-like arrays is a matrix multiplication

```
>>> print(CArray([10,20]).dot(CArray([1,0], tosparse=True).T))
10
>>> print(CArray([10,20], tosparse=True).dot(CArray([1,0]).T))
10
```

property dtype
Data-type of stored data.

classmethod empty(*shape*, *dtype=<class 'float'>*, *sparse=False*)
Return a new array of given shape and type, without filling it.

Parameters

shape [int or tuple] Shape of the new array, e.g., 2 or (2,3).

dtype [str or dtype, optional] The desired data-type for the array. Default is float.

sparse [bool, optional] If False (default) a dense array will be returned. Otherwise, a sparse array is returned.

Returns

CArray Array of arbitrary values with the given properties.

Notes

.empty, unlike *.zeros*, does not set the array values to zero, and may therefore be marginally faster. On the other hand, it requires the user to manually set all the values in the array, and should be used with caution.

Examples

```
>>> from secml.array import CArray
```

```
>>> array = CArray.empty(3)
>>> print(array
CArray([  0.00000000e+000   4.94944794e+173   6.93660640e-310])  # random
```

```
>>> array = CArray.empty((2,1), dtype=int, sparse=True)
>>> print(array)
CArray()
>>> print(array.shape)
(2, 1)
```

exp (*self*)

Calculate the exponential of all elements in the input array.

DENSE FORMAT ONLY

Returns

CArray New array with element-wise exponential of current data.

Notes

The irrational number e is also known as Euler's number. It is approximately 2.718281, and is the base of the natural logarithm, `ln` (this means that, if $x = \ln y = \log_e y$, then $e^x = y$. For real input, `exp(x)` is always positive.

For complex arguments, `x = a + ib`, we can write $e^x = e^a e^{ib}$. The first term, e^a, is already known (it is the real argument, described above). The second term, e^{ib}, is $\cos b + i \sin b$, a function with magnitude 1 and a periodic phase.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([0,1,3]).exp())
CArray([ 1.         2.718282 20.085537])
```

classmethod eye (*n_rows*, *n_cols=None*, *k=0*, *dtype=<class 'float'>*, *sparse=False*)

Return a 2-D array with ones on the diagonal and zeros elsewhere.

Parameters

n_rows [int] Number of rows in the output.

n_cols [int or None, optional] Number of columns in the output. If None, defaults to n_rows.

k [int, optional] Index of the diagonal: 0 (the default) refers to the main diagonal, a positive value refers to an upper diagonal, and a negative value to a lower diagonal.

dtype [str or dtype, optional] Data-type of the returned array.

sparse [bool, optional] If False (default) a dense array will be returned. Otherwise, a sparse array will be created.

Returns

CArray An array where all elements are equal to zero, except for the k-th diagonal, whose values are equal to one.

Examples

```
>>> from secml.array import CArray
>>> array = CArray.eye(2)
>>> print(array)
CArray([[1. 0.]
 [0. 1.]])
```

```
>>> array = CArray.eye(2, 3, k=1, dtype=int, sparse=True)
>>> print(array)
CArray(  (0, 1) 1
  (1, 2)        1)
```

```
>>> print(array.shape)
(2, 3)
```

find(*self*, *condition*)

Returns vector-like array elements indices depending on condition.

Parameters

condition [CArray] Array with booleans representing desired condition.

Returns

list List with indices corresponding to array elements where condition is True.

See also:

find_2d find method for arrays of generic shape.

Examples

```
>>> from secml.array import CArray
```

```
>>> array = CArray([1,0,-6,2,0])
>>> array_find = array.find(array > 0)
>>> print(array_find)
[0, 3]
>>> print(array[array_find])
CArray([1 2])
```

```
>>> array = CArray([[1,0,-6,2,0]])
>>> array_find = array.find(array == 0)
>>> print(array_find)
[1, 4]
>>> print(array[array_find].shape)
(1, 2)
```

```
>>> array = CArray([[1,0,-6,2,0]], tosparse=True)
>>> array_find = array.find(array == 0)
```

(continues on next page)

(continued from previous page)

```
>>> print(array_find)
[1, 4]
>>> print(array[array_find].shape)
(1, 2)
```

find_2d(*self*, *condition*)

Returns array elements indices depending on condition.

Parameters

condition [CArray] Array with booleans representing desired condition.

Returns

list List of len(out_find) == ndim with indices corresponding to array elements where condition is True. Es. for matrices, out_find[0] holds the indices of rows, out_find[1] the indices of columns.

Notes

Using .find_2d() output for indexing original array always result in a ravelled array with elements which corresponding condition was True.

Examples

```
>>> from secml.array import CArray
```

```
>>> array = CArray([[1,0],[-6,3],[2,7]])
>>> array_find = array.find_2d(array > 0)
>>> print(array_find)
[[0, 1, 2, 2], [0, 1, 0, 1]]
>>> print(array[array_find])
CArray([1 3 2 7])
```

```
>>> array = CArray([[1,0],[-6,0],[2,0]], tosparse=True)
>>> array_find = array.find_2d(array == 0)
>>> print(array_find)
[[0, 1, 2], [1, 1, 1]]
>>> print(array[array_find].shape)
(1, 3)
```

```
>>> array = CArray([1,0,2])
>>> array_find = array.find_2d(array > 0)
>>> print(array_find)
[[0, 0], [0, 2]]
>>> print(array[array_find])
CArray([1 2])
```

flatten(*self*)

Return a flattened copy of array.

For dense format a 1-dim array, containing the elements of the input, is returned. For sparse format a (1 x array.size) array will be returned.

Returns

CArray Output of the same dtype as a, of shape (array.size,) for dense format or (1,array.size) for sparse format.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([[1,2],[3,4]]).flatten())
CArray([1 2 3 4])
```

```
>>> print(CArray([[1],[2],[3]], tosparse=True).flatten())
CArray(  (0, 0) 1
  (0, 1)        2
  (0, 2)        3)
```

floor (*self*)

Return the floor of the input, element-wise.

The floor of the scalar x is the largest integer i, such that i <= x.

Returns

out_floor [CArray] The floor of each element in x, with float dtype.

See also:

CArray.round Evenly round to the given number of decimals.

CArray.ceil Return the ceiling of the input, element-wise.

Notes

Some spreadsheet programs calculate the "floor-towards-zero", in other words floor(-2.5) == -2. We instead uses the definition of floor where floor(-2.5) == -3.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([-1.7, -1.5, -0.2, 0.2, 1.5, 1.7, 2.0]).floor())
CArray([-2. -2. -1.  0.  1.  1.  2.])
```

```
>>> # Array with dtype == int is upcasted to float before flooring
>>> print(CArray([[-2, -1], [1, 1]], tosparse=True).floor())
CArray(  (0, 0) -2.0
  (0, 1)        -1.0
  (1, 0)        1.0
  (1, 1)        1.0)
```

classmethod from_iterables (*iterables_list*)

Build an array by chaining elements from objects in the input list.

Parameters

iterables_list [list of iterable] List of iterables to chain. Valid objects are CArrays, lists, tuples, and any other iterable. N-Dimensional arrays are flattened before chaining.

Returns

CArray Flat CArray with all values chained from input objects.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray.from_iterables([[1, 2], (3, 4), CArray([5, 6])]))
CArray([1 2 3 4 5 6])
```

```
>>> print(CArray.from_iterables([CArray([1, 2]), CArray([[3, 4], [5, 6]])]))
CArray([1 2 3 4 5 6])
```

get_data(*self*)
Return stored data as a standard array type.

Returns

np.ndarray or scipy.sparse.csr_matrix If array is dense, a np.ndarray is returned. If array is sparse, a scipy.sparse.csr_matrix is returned.

See also:

tondarray returns a np.ndarray, regardless of array format.

tocsr returns a scipy.sparse.csr_matrix, regardless of array format.

get_nnz(*self*, *axis=None*)
Counts the number of non-zero values in the array.

Parameters

axis [bool or None, optional] Axis or tuple of axes along which to count non-zeros. Default is None, meaning that non-zeros will be counted along a flattened version of the array.

Returns

count [CArray or int] Number of non-zero values in the array along a given axis. Otherwise, the total number of non-zero values in the array is returned.

Examples

```
>>> from secml.array import CArray
```

```
>>> a = CArray([[1,2],[0,5],[0,0],[2,0]])
>>> print(a.get_nnz())  # Total number of non-zero elements
4
>>> print(a.get_nnz(axis=0))  # Number of non-zero elements for each column
CArray([2 2])
>>> print(a.get_nnz(axis=1))  # Number of non-zero elements for each row
CArray([2 1 0 1])
```

has_compatible_shape(*self*, *other*)
Return True if input CArray has a compatible shape.

Two CArrays can be considered compatible if both have the same shape or both are vector-like.

Parameters

other [CArray] Array to check for shape compatibility

Returns

bool True if input array has compatible shape with current array.

See also:

is_vector_like check if an array is vector-like.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([[1,2]]).has_compatible_shape(CArray([[1],[2]])))
False
```

```
>>> print(CArray([1,2]).has_compatible_shape(CArray([1,2,3])))
False
```

```
>>> print(CArray([[1,2]], tosparse=True).has_compatible_shape(CArray([1,2])))
True
```

interp(*self*, *x_data*, *y_data*, *return_left=None*, *return_right=None*)
One-dimensional linear interpolation.

DENSE FORMAT ONLY

Returns the 1-D piecewise linear interpolant to a function with given values at discrete data-points.

Parameters

x_data [CArray] Flat array of floats with the x-coordinates of the data points, must be increasing.

y_data [CArray] Flat array of floats with the y-coordinates of the data points, same length as *x_data*.

return_left [float, optional] Value to return for x < x_data[0], default is y_data[0].

return_right [float, optional] Value to return for x > x_data[-1], default is y_data[-1].

Returns

CArray The interpolated values, same shape as x.

Notes

The function does not check that the x-coordinate sequence *x_data* is increasing. If *x_data* is not increasing, the results are nonsense.

Examples

```
>>> from secml.array import CArray
>>> from secml.figure import CFigure
>>> from secml.core.constants import pi
```

```
>>> fig = CFigure(fontsize=14)
>>> x_array = CArray.linspace(0, 2*pi, 10)
>>> y_array = x_array.sin()
>>> x_vals = CArray.linspace(0, 2*pi, 50)
```

```
>>> y_interp = x_vals.interp(x_array, y_array)
```

```
>>> fig.sp.plot(x_array, y_array, 'o')
>>> fig.sp.plot(x_vals, y_interp, '-xr')
```

inv(*self*)

Compute the (multiplicative) inverse of a square matrix.

Given a square matrix a, return the matrix inv satisfying dot(array, array_inv) = dot(array_inv, array) = eye(array.shape[0]).

Returns

array_inv [CArray] (Multiplicative) inverse of the square matrix.

Raises

LinAlgError [dense only] If array is not square or inversion fails.

ValueError [sparse only] If array is not square or inversion fails

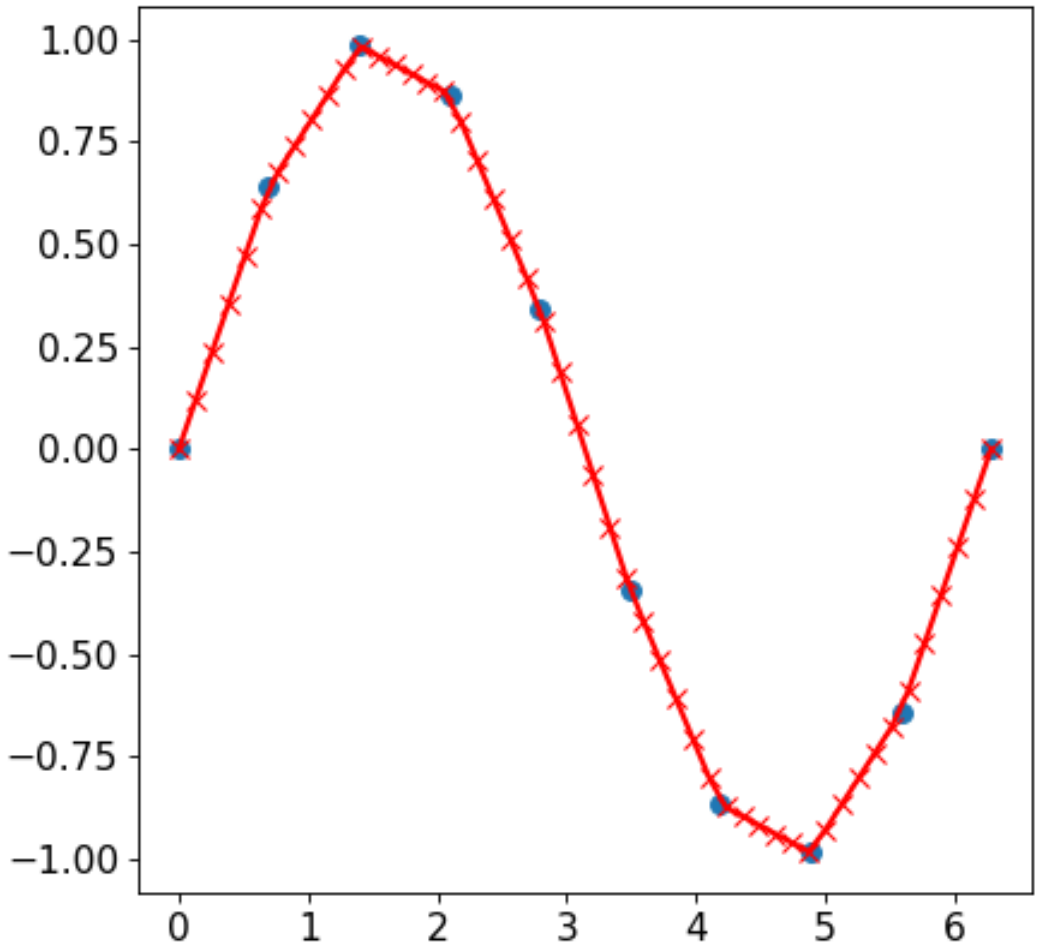
Notes

If the inverse of a sparse array is expected to be non-sparse, it will likely be faster to convert array to dense first.

Examples

```
>>> from secml.array import CArray
```

```
>>> array = CArray([[1., 2.], [3., 4.]])
>>> array_inv = array.inv()
>>> (array.dot(array_inv).round() == CArray.eye(2)).all()
True
>>> (array_inv.dot(array).round() == CArray.eye(2)).all()
True
```


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
0
1
2
3
4
5
6

```
>>> print(CArray([[1., 2.], [3., 4.]], tosparse=True).inv().round(1))
CArray(  (0, 0) -2.0
  (0, 1)         1.0
  (1, 0)         1.5
  (1, 1)         -0.5)
```

```
>>> CArray([[1.,2.,3.], [4., 5.,6.]]).inv()
Traceback (most recent call last):
    ...
numpy.linalg.LinAlgError: Last 2 dimensions of the array must be square
```

is_inf (*self*)

Test element-wise for positive or negative infinity.

Returns

CArray Array of the same shape as x, with True where x == +/-inf, otherwise False.

Examples

```
>>> from secml.core.constants import inf, nan
>>> from secml.array import CArray
```

```
>>> a = CArray([1, inf, -inf, nan, 4.5])
>>> print(a.is_inf())
CArray([False  True  True False False])
```

is_nan (*self*)

Test element-wise for Not a Number (NaN).

Returns

CArray Array of the same shape as x, with True where x == nan, otherwise False.

Examples

```
>>> from secml.core.constants import inf, nan
>>> from secml.array import CArray
```

```
>>> a = CArray([1, inf, -inf, nan, 4.5])
>>> print(a.is_nan())
CArray([False False False  True False])
```

is_neginf (*self*)

Test element-wise for negative infinity.

Returns

CArray Array of the same shape as x, with True where x == -inf, otherwise False.

Examples

```
>>> from secml.core.constants import inf, nan
>>> from secml.array import CArray
```

```
>>> a = CArray([1, inf, -inf, nan, 4.5])
>>> print(a.is_neginf())
CArray([False False  True False False])
```

is_posinf(*self*)

Test element-wise for positive infinity.

Returns

CArray Array of the same shape as x, with True where x == +inf, otherwise False.

Examples

```
>>> from secml.core.constants import inf, nan
>>> from secml.array import CArray
```

```
>>> a = CArray([1, inf, -inf, nan, 4.5])
>>> print(a.is_posinf())
CArray([False  True False False False])
```

property is_vector_like

True if array is vector-like.

An array is vector-like when 1-Dimensional or 2-Dimensional with shape[0] == 1.

Returns

bool True if array is vector-like.

Examples

```
>>> from secml.array import CArray
```

```
>>> a = CArray([1,2,3])
>>> a.is_vector_like
True
```

```
>>> a = CArray([1,2,3], tosparse=True)  # sparse arrays always 2-D
>>> a.is_vector_like
True
```

```
>>> a = CArray([[1,2],[3,4]])
>>> a.is_vector_like
False
```

property isdense

True if data is stored in DENSE form, False otherwise.

Returns

bool True if data is stored in DENSE form, False otherwise.

property issparse
True if data is stored in SPARSE form, False otherwise.

Returns

bool True if data is stored in SPARSE form, False otherwise.

item(*self*)
Returns the single element in the array as built-in type.

Returns

int, float, bool, str The single element in the array.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([1]).item())
1
```

```
>>> print(CArray([[1.]]).item())
1.0
```

```
>>> print(CArray([1], tosparse=True).item())
1
```

```
>>> print(CArray([1,2,3]).item())
Traceback (most recent call last):
    ...
ValueError: cannot use .item(). Array has size 3
```

```
>>> print(CArray([]).item())
Traceback (most recent call last):
    ...
ValueError: cannot use .item(). Array has size 0
```

classmethod linspace(*start*, *stop*, *num=50*, *endpoint=True*, *sparse=False*)
Return evenly spaced numbers over a specified interval.

Returns num evenly spaced float samples, calculated over the interval [start, stop]. The endpoint of the interval can optionally be excluded.

Parameters

start [scalar] The starting value of the sequence.

stop [scalar] The end value of the sequence, unless endpoint is set to False. In that case, the sequence consists of all but the last of num + 1 evenly spaced samples, so that stop is excluded. Note that the step size changes when endpoint is False.

num [int, optional] Number of samples to generate. Default is 50.

endpoint [bool, optional] If True, stop is the last sample. Otherwise, it is not included. Default is True.

sparse [bool, optional] If False (default) a dense array will be returned. Otherwise, a sparse array is created.

Returns

CArray There are num equally spaced samples in the closed interval [start, stop] or the half-open interval [start, stop) (depending on whether endpoint is True or False).

Warning: When sparse is True, array is created as dense and then converted to sparse format. Consequently, the performance of this method is not comparable to other sparse array creation routines.

See also:

CArray.arange Similar to linspace, but uses a specific step size.

Examples

```
>>> from secml.array import CArray
```

```
>>> array = CArray.linspace(3.0, 4, num=5)
>>> print(array)
CArray([3.   3.25 3.5  3.75 4.  ])
```

```
>>> array = CArray.linspace(3, 4., num=5, endpoint=False)
>>> print(array)
CArray([3.  3.2 3.4 3.6 3.8])
```

classmethod load(*datafile*, *dtype=<class 'float'>*, *arrayformat='dense'*, *startrow=0*, *skipend=0*, *cols=None*)

Load array data from plain text file.

The default encoding is *utf-8*.

Parameters

datafile [str or file_handle] File or filename to read. If the filename extension is gz or bz2, the file is first decompressed.

dtype [str, dtype, optional] Data type of the resulting array, default 'float'. If None, the dtype will be determined by the contents of the file.

arrayformat [{'dense', 'sparse'}, optional] Format of array to load, default 'dense'.

startrow [int, optional, dense only] Array row to start loading from.

skipend [int, optional, dense only] Number of lines to skip from the end of the file when reading.

cols [{CArray, int, tuple}, optional, dense only] Columns to load from target file.

Returns

CArray Array resulting from loading, 2-Dimensional.

log(*self*)

Calculate the natural logarithm of all elements in the input array.

DENSE FORMAT ONLY

Returns

CArray New array with element-wise natural logarithm of current data.

Notes

Logarithm is a multivalued function: for each *x* there is an infinite number of *z* such that *exp(z) = x*. The convention is to return the *z* whose imaginary part lies in *[-pi, pi]*.

For real-valued input data types, *log* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *log* is a complex analytical function that has a branch cut *[-inf, 0]* and is continuous from above on it. *log* handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

References

[1], [2]

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([0,1,3]).log())
CArray([    -inf 0.       1.098612])
```

log10(*self*)

Calculate the base 10 logarithm of all elements in the input array.

DENSE FORMAT ONLY

Returns

CArray New array with element-wise base 10 logarithm of current data.

Notes

Logarithm is a multivalued function: for each *x* there is an infinite number of *z* such that *10**z = x*. The convention is to return the *z* whose imaginary part lies in *[-pi, pi]*.

For real-valued input data types, *log10* always returns real output. For each value that cannot be expressed as a real number or infinity, it yields `nan` and sets the *invalid* floating point error flag.

For complex-valued input, *log10* is a complex analytical function that has a branch cut *[-inf, 0]* and is continuous from above on it. *log10* handles the floating-point negative zero as an infinitesimal negative number, conforming to the C99 standard.

References

[1], [2]

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([0,1,3]).log10())
CArray([    -inf 0.       0.477121])
```

logical_and(*self*, *array*)
Element-wise logical AND of array elements.

Compare two arrays and returns a new array containing the element-wise logical AND.

Parameters

array [CArray] The array holding the elements to compare current array with. Must have the same shape of first array.

Returns

CArray The element-wise logical AND between the two arrays. If one of the two arrays is sparse, result will be sparse.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([[-1,0],[2,0]]).logical_and(CArray([[2,-1],[2,-1]])))
CArray([[ True False]
 [ True False]])
```

```
>>> print(CArray([-1]).logical_and(CArray([2])))
CArray([ True])
```

```
>>> array = CArray([1,0,2,-1])
>>> print((array > 0).logical_and(array < 2))
CArray([ True False False False])
```

logical_not(*self*)
Element-wise logical NOT of array elements.

Returns

CArray The element-wise logical NOT.

Notes

For sparse arrays this operation is usually really expensive as the number of zero elements is higher than the number of non-zeros.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([[-1,0],[2,0]]).logical_not())
CArray([[False  True]
 [False  True]])
```

```
>>> print(CArray([True]).logical_not())
CArray([False])
```

```
>>> array = CArray([1,0,2,-1])
>>> print((array > 0).logical_not())
CArray([False  True False  True])
```

logical_or(*self*, *array*)

Element-wise logical OR of array elements.

Compare two arrays and returns a new array containing the element-wise logical OR.

Parameters

array [CArray or array_like] The array like object holding the elements to compare current array with. Must have the same shape of first array.

Returns

out_and [CArray] The element-wise logical OR between the two arrays.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([[-1,0],[2,0]]).logical_or(CArray([[2,0],[2,-1]])))
CArray([[ True False]
 [ True  True]])
```

```
>>> print(CArray([True]).logical_or(CArray([False])))
CArray([ True])
```

```
>>> array = CArray([1,0,2,-1])
>>> print((array > 0).logical_or(array < 2))
CArray([ True  True  True  True])
```

max(*self*, *axis=None*, *keepdims=True*)

Return the maximum of an array or maximum along an axis.

Parameters

axis [int or None, optional] Axis along which to operate. If None (default), array is flattened before use.

keepdims [bool, optional] If this is set to True (default), the result will broadcast correctly against the original array. Otherwise resulting array is flattened.

Returns

scalar or CArray Maximum of the array. If axis is None, scalar is returned. Otherwise, a CArray with shape and number of dimensions consistent with the original array and the axis parameter is returned.

Notes

- **For sparse arrays all elements are taken into account (both** zeros and non-zeros).
- **NaN values are propagated, that is if at least one item is NaN,** the corresponding max value will be NaN as well.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([[-1,0],[2,0]], tosparse=True).max())
2
```

```
>>> print(CArray([[-1,0],[2,0]]).max(axis=0))
CArray([[2 0]])
>>> print(CArray([[-1,0],[2,0]]).max(axis=1))
CArray([[0]
 [2]])
```

```
>>> print(CArray([-1,0,2,0]).max(axis=0))
CArray([-1  0  2  0])
>>> print(CArray([-1,0,2,0]).max(axis=1))
CArray([2])
```

```
>>> from secml.core.constants import nan
>>> print(CArray([5,nan]).max())
nan
```

maximum (*self*, *array*)

Element-wise maximum of array elements.

Compare two arrays and returns a new array containing the element-wise maximum. If one of the elements being compared is a NaN, then that element is returned. If both elements are NaNs then the first is returned. The latter distinction is important for complex NaNs, which are defined as at least one of the real or imaginary parts being a NaN. The net effect is that NaNs are propagated.

Parameters

array [CArray or array_like] The array like object holding the elements to compare current array with. Must have the same shape of first array.

Returns

CArray The element-wise maximum between the two arrays.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([[-1,0],[2,0]]).maximum(CArray([[2,-1],[2,-1]])))
CArray([[2 0]
 [2 0]])
```

```
>>> print(CArray([[-1,0],[2,0]], tosparse=True).maximum(CArray([[2,-1],[2,-
↪1]])))
CArray(  (0, 0) 2
  (1, 0)        2)
```

```
>>> print(CArray([-1]).maximum(CArray([2])))
CArray([2])
```

mean (*self*, *axis=None*, *dtype=None*, *keepdims=True*)

Compute the arithmetic mean along the specified axis.

Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the specified axis. Output is casted to dtype float.

Parameters

axis [int, optional] Axis along which the means are computed. The default is to compute the mean of the flattened array.

dtype [data-type, optional] Type to use in computing the mean. For integer inputs, the default is float64; for floating point inputs, it is the same as the input dtype.

keepdims [bool, optional] If this is set to True (default), the result will broadcast correctly against the original array.

Returns

float or CArray Mean of the elements in the array. If axis is None, float is returned. Otherwise, a CArray with shape and number of dimensions consistent with the original array and the axis parameter is returned.

Notes

The arithmetic mean is the sum of the elements along the axis divided by the number of elements.

Note that for floating-point input, the mean is computed using the same precision the input has. Depending on the input data, this can cause the results to be inaccurate, especially for float32 (see example below). Specifying a higher-precision accumulator using the dtype keyword can alleviate this issue.

By default, float16 results are computed using float32 intermediates for extra precision.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([[1,4],[4,3]], tosparse=True).mean())
3.0
```

```
>>> print(CArray([[1,4],[4,3]], tosparse=True).mean(axis=0))
CArray([[2.5 3.5]])
```

```
>>> print(CArray([1,4,4,3]).mean(axis=0))
CArray([1. 4. 4. 3.])
>>> print(CArray([1,4,4,3]).mean(axis=1))
CArray([3.])
```

median (*self*, *axis=None*, *keepdims=True*)

Compute the median along the specified axis.

Given a vector V of length N, the median of V is the middle value of a sorted copy of V, V_sorted - i e., V_sorted[(N-1)/2], when N is odd, and the average of the two middle values of V_sorted when N is even.

DENSE FORMAT ONLY

Parameters

axis [int, optional] Axis along which the means are computed. The default is to compute the median of the flattened array.

keepdims [bool, optional] If this is set to True (default), the result will broadcast correctly against the original array.

Returns

float or CArray Median of the elements in the array. If axis is None, float is returned. Otherwise, a CArray with shape and number of dimensions consistent with the original array and the axis parameter is returned.

Notes

If the input contains integers or floats smaller than float64, then the output data-type is np.float64. Otherwise, the data-type of the output is the same as that of the input.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([[1,4],[4,3]]).median())
3.5
```

```
>>> print(CArray([[1,4],[4,3]]).median(axis=0))
CArray([[2.5 3.5]])
```

```
>>> print(CArray([1,4,3]).median())  # array size is odd
3.0
```

```
>>> print(CArray([1,4,4,3]).median(axis=0))
CArray([1. 4. 4. 3.])
>>> print(CArray([1,4,4,3]).median(axis=1))
CArray([3.5])
```

classmethod meshgrid(*xi*, *indexing='xy'*)

Return coordinate matrices from coordinate vectors.

DENSE ARRAYS ONLY

Make N-D coordinate arrays for vectorized evaluations of N-D scalar/vector fields over N-D grids, given one-dimensional coordinate arrays x1, x2,..., xn.

Parameters

x1, x2, ..., xi [tuple of CArray or list] 1-D arrays representing the coordinates of a grid.

indexing [{'xy', 'ij'}, optional] Cartesian ('xy', default) or matrix ('ij') indexing of output. See Examples.

Returns

X1, X2, ..., XN [tuple of CArray] For vectors x1, x2,..., 'xn' with lengths Ni=len(xi), return (N1, N2, N3,...Nn) shaped arrays if indexing='ij' or (N2, N1, N3,...Nn) shaped arrays if indexing='xy' with the elements of xi repeated to fill the matrix along the first dimension for x1, the second for x2 and so on.

Examples

```
>>> from secml.array import CArray
```

```
>>> x = CArray([1,3,5])
>>> y = CArray([2,4,6])
>>> xv, yv = CArray.meshgrid((x, y))
>>> print(xv)
CArray([[1 3 5]
 [1 3 5]
 [1 3 5]])
>>> print(yv)
CArray([[2 2 2]
 [4 4 4]
 [6 6 6]])
```

```
>>> xv, yv = CArray.meshgrid((x, y), indexing='ij')
>>> print(xv)
CArray([[1 1 1]
 [3 3 3]
 [5 5 5]])
>>> print(yv)
CArray([[2 4 6]
 [2 4 6]
 [2 4 6]])
```

min(*self*, *axis=None*, *keepdims=True*)

Return the minimum of an array or minimum along an axis.

Parameters

axis [int or None, optional] Axis along which to operate. If None (default), array is flattened before use.

keepdims [bool, optional] If this is set to True (default), the result will broadcast correctly against the original array. Otherwise resulting array is flattened.

Returns

scalar or CArray Minimum of the array. If axis is None, scalar is returned. Otherwise, a CArray with shape and number of dimensions consistent with the original array and the axis parameter is returned.

Notes

- **For sparse arrays all elements are taken into account (both** zeros and non-zeros).
- **NaN values are propagated, that is if at least one item is NaN,** the corresponding max value will be NaN as well.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([[-1,0],[2,0]], tosparse=True).min())
-1
```

```
>>> print(CArray([[-2,0],[-1,0]]).min(axis=0))
CArray([[-2  0]])
>>> print(CArray([[-2,0],[-1,0]]).min(axis=1))
CArray([[-2]
 [-1]])
```

```
>>> print(CArray([-1,0,2,0]).min(axis=0))
CArray([-1  0  2  0])
>>> print(CArray([-1,0,2,0]).min(axis=1))
CArray([-1])
```

```
>>> from secml.core.constants import nan
>>> print(CArray([5,nan]).min())
nan
```

minimum(*self*, *array*)

Element-wise minimum of array elements.

Compare two arrays and returns a new array containing the element-wise minimum. If one of the elements being compared is a NaN, then that element is returned. If both elements are NaNs then the first is returned. The latter distinction is important for complex NaNs, which are defined as at least one of the real or imaginary parts being a NaN. The net effect is that NaNs are propagated.

Parameters

array [CArray or array_like] The array like object holding the elements to compare current array with. Must have the same shape of first array.

Returns

CArray The element-wise minimum between the two arrays.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([[-1,0],[2,0]]).minimum(CArray([[2,-1],[2,-1]])))
CArray([[-1 -1]
 [ 2 -1]])
```

```
>>> print(CArray([[-1,0],[2,0]], tosparse=True).minimum(CArray([[2,-1],[2,-
↪1]])))
CArray(  (0, 0) -1
  (0, 1)        -1
  (1, 0)        2
  (1, 1)        -1)
```

```
>>> print(CArray([-1]).minimum(CArray([2])))
CArray([-1])
```

nan_to_num(*self*)

Replace nan with zero and inf with finite numbers.

Replace array elements if Not a Number (NaN) with zero, if (positive or negative) infinity with the largest (smallest or most negative) floating point value that fits in the array dtype. All finite numbers are upcast to the output dtype (default float64).

Notes

We use the IEEE Standard for Binary Floating-Point for Arithmetic (IEEE 754). This means that Not a Number is not equivalent to infinity.

Examples

```
>>> from secml.array import CArray
>>> from secml.core.constants import nan, inf
>>> import numpy as np
>>> np.set_printoptions(precision=1)
```

```
>>> array = CArray([-1,0,1,nan,inf,-inf])
>>> array.nan_to_num()
>>> print(array)
CArray([-1.000000e+000  0.000000e+000  1.000000e+000  0.000000e+000  1.
↪797693e+308
 -1.797693e+308])
```

```
>>> # Restoring default print precision
>>> np.set_printoptions(precision=8)
```

nanargmax(*self*, *axis=None*)

Indices of the maximum values along an axis ignoring NaNs.

For all-NaN slices ValueError is raised. Warning: the results cannot be trusted if a slice contains only NaNs and infs.

DENSE ARRAYS ONLY

Parameters

axis [int, None, optional] If None (default), array is flattened before computing index, otherwise the specified axis is used.

Returns

int or CArray Index of the maximum of the array ignoring NaNs. If axis is None, int is returned. Otherwise, a CArray with shape and number of dimensions consistent with the original array and the axis parameter is returned.

Notes

In case of multiple occurrences of the maximum values, the indices corresponding to the first occurrence are returned.

Examples

```
>>> from secml.array import CArray
```

```
>>> from secml.core.constants import nan
>>> print(CArray([5, nan]).argmax())
1
```

```
>>> print(CArray([5, nan]).nanargmax())
0
```

```
>>> print(CArray([[-1, nan], [nan, 0]]).nanargmax())
3
```

```
>>> print(CArray([[-1, nan], [nan, 0]]).nanargmax(axis=0))
CArray([[0 1]])
>>> print(CArray([[-1, nan], [nan, 0]]).nanargmax(axis=1))
CArray([[0]
 [1]])
```

nanargmin (*self*, *axis=None*)

Indices of the minimum values along an axis ignoring NaNs

For all-NaN slices ValueError is raised. Warning: the results cannot be trusted if a slice contains only NaNs and infs.

Parameters

axis [int, None, optional] If None (default), array is flattened before computing index, otherwise the specified axis is used.

Returns

int or CArray Index of the minimum of the array ignoring NaNs. If axis is None, int is returned. Otherwise, a CArray with shape and number of dimensions consistent with the original array and the axis parameter is returned.

Notes

In case of multiple occurrences of the minimum values, the indices corresponding to the first occurrence are returned.

Examples

```
>>> from secml.array import CArray
```

```
>>> from secml.core.constants import nan
>>> print(CArray([5, nan]).argmin())
1
```

```
>>> print(CArray([5, nan]).nanargmin())
0
```

```
>>> print(CArray([[-1, nan], [nan, 0]]).nanargmin())
0
```

```
>>> print(CArray([[-1, nan], [nan, 0]]).nanargmin(axis=0))
CArray([[0 1]])
>>> print(CArray([[-1, nan], [nan, 0]]).nanargmin(axis=1))
CArray([[0]
 [1]])
```

nanmax (*self*, *axis=None*, *keepdims=True*)

Return the maximum of an array or maximum along an axis ignoring NaNs.

When all-NaN slices are encountered a RuntimeWarning is raised and Nan is returned for that slice.

DENSE ARRAYS ONLY

Parameters

axis [int or None, optional] Axis along which to operate. If None (default), flattened input is used.

keepdims [bool, optional, dense only] If this is set to True (default), the result will broadcast correctly against the original array. Otherwise resulting array is flattened.

Returns

scalar or CArray Maximum of the array ignoring NaNs. If axis is None, scalar is returned. Otherwise, a CArray with shape and number of dimensions consistent with the original array and the axis parameter is returned.

Examples

```
>>> from secml.array import CArray
```

```
>>> from secml.core.constants import nan
>>> print(CArray([5, nan]).max())
nan
```

```
>>> print(CArray([5, nan]).nanmax())
5.0
```

```
>>> print(CArray([[-1, nan], [nan, 0]]).nanmax())
0.0
```

```
>>> print(CArray([[-1, nan], [nan, 0]]).nanmax(axis=0))
CArray([[-1.  0.]])
>>> print(CArray([[-1, nan], [nan, 0]]).nanmax(axis=1))
CArray([[-1.]
 [ 0.]])
```

nanmin (*self*, *axis=None*, *keepdims=True*)

Return the minimum of an array or minimum along an axis ignoring NaNs.

When all-NaN slices are encountered a RuntimeWarning is raised and Nan is returned for that slice.

DENSE ARRAYS ONLY

Parameters

axis [int or None, optional] Axis along which to operate. If None (default), flattened input is used.

keepdims [bool, optional, dense only] If this is set to True (default), the result will broadcast correctly against the original array. Otherwise resulting array is flattened.

Returns

scalar or CArray Index of the minimum of the array ignoring NaNs. If axis is None, scalar is returned. Otherwise, a CArray with shape and number of dimensions consistent with the original array and the axis parameter is returned.

Examples

```
>>> from secml.array import CArray
```

```
>>> from secml.core.constants import nan
>>> print(CArray([5, nan]).min())
nan
```

```
>>> print(CArray([5, nan]).nanmin())
5.0
```

```
>>> print(CArray([[-1, nan], [nan, 0]]).nanmin())
-1.0
```

```
>>> print(CArray([[-1, nan], [nan, 0]]).nanmin(axis=0))
CArray([[-1.  0.]])
>>> print(CArray([[-1, nan], [nan, 0]]).nanmin(axis=1))
CArray([[-1.]
 [ 0.]])
```

property ndim

Number of array dimensions.

This is always 2 for sparse arrays.

property nnz

Number of non-zero values in the array.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([1,0,3,0], tosparse=True).nnz)
2
```

property nnz_data

Return non-zero array elements.

Returns

nnz_data [CArray] Flat array, dense, shape (n,), with non-zero array elements.

Examples

```
>>> from secml.array import CArray
```

```
>>> array = CArray([1,0,3,0], tosparse=True)
>>> print(array.nnz_data)
CArray([1 3])
```

property nnz_indices

Index of non-zero array elements.

Returns

nnz_indices [list] List of 2 lists. Inside out[0] there are the indices of the corresponding rows and inside out[1] there are the indices of the corresponding columns of non-zero array elements.

Examples

```
>>> from secml.array import CArray
```

```
>>> array = CArray([1,0,3,0], tosparse=True)
>>> nzz_indices = array.nnz_indices
>>> nzz_indices
[[0, 0], [0, 2]]
>>> print(array[nzz_indices])
CArray(  (0, 0) 1
  (0, 1)        3)
```

```
>>> array = CArray([1,0,3,0])
>>> nzz_indices = array.nnz_indices
>>> nzz_indices
[[0, 0], [0, 2]]
>>> print(array[nzz_indices])
CArray([1 3])
```

norm(*self*, *order=None*)

Entrywise vector norm.

This function provides vector norms on vector-like arrays.

This function is able to return one of an infinite number of vector norms (described below), depending on the value of the order parameter.

Parameters

order [{int, inf, -inf}, optional] Order of the norm (see table under Notes).

Returns

float Norm of the array.

See also:

numpy.norm Full description of different norms.

Notes

For integer order parameter, norm is computed as norm = sum(abs(array)**order)**(1./order). For other norm types, see np.norm description.

Negative vector norms are only supported for dense arrays.

Differently from numpy, we consider flat vectors as 2-Dimensional with shape (1,array.size).

If input 2-Dimensional array is NOT vector-like, ValueError will be raised.

Examples

```
>>> from secml.array import CArray
>>> from secml.core.constants import inf
```

```
>>> print(round(CArray([1,2,3]).norm(), 5))
3.74166
>>> print(round(CArray([[1,2,3]]).norm(2), 5))
3.74166
```

```
>>> print(CArray([1,2,3]).norm(1))
6.0
>>> print(CArray([1,2,3]).tosparse().norm(1))
6.0
```

```
>>> print(CArray([1,2,3]).norm(inf))
3.0
>>> print(CArray([1,2,3]).norm(-inf))
1.0
```

```
>>> print(CArray([[1,2],[2,4]]).norm())
Traceback (most recent call last):
    ...
ValueError: Array has shape (2, 2). Call .norm_2d() to compute matricial norm␣
↪or vector norm along axis.
```

norm_2d(*self*, *order=None*, *axis=None*, *keepdims=True*)

Matrix norm or vector norm along axis.

This function provides matrix norm or vector norm along axis of 2D arrays. Flat arrays will be converted to 2D before computing the norms.

This function is able to return one of seven different matrix norms, or one of an infinite number of vector norms (described below), depending on the value of the order parameter.

Parameters

order [{'fro', non-zero int, inf, -inf}, optional] Order of the norm (see table under Notes). 'fro' stands for Frobenius norm.

axis [int or None, optional] If axis is an integer, it specifies the axis of array along which to compute the vector norms. If axis is None then the matrix norm is returned.

keepdims [bool, optional] If this is set to True (default), the result will broadcast correctly against the original array. Otherwise resulting array is flattened.

Returns

float or CArray Norm of the array. If axis is None, float is returned. Otherwise, a CArray with shape and number of dimensions consistent with the original array and the axis parameter is returned.

See also:

numpy.norm Full description of different norms.

Notes

For integer order parameter, norm is computed as norm = sum(abs(array)**order)**(1./order). For other norm types, see np.norm description. Negative vector norms along axis are only supported for dense arrays.

Examples

```
>>> from secml.array import CArray
>>> from secml.core.constants import inf
```

```
>>> print(round(CArray([1,2,3]).norm_2d(), 5))
3.74166
```

```
>>> print(CArray([1,2,3]).norm_2d(1))  # max(sum(abs(x), axis=0))
3.0
>>> print(CArray([[1,2,3]]).norm_2d(1))
3.0
```

```
>>> print(CArray([1,2,3]).norm_2d(inf))  # max(sum(abs(x), axis=1))
6.0
>>> print(CArray([1,2,3]).norm_2d(-inf))  # min(sum(abs(x), axis=1))
6.0
```

```
>>> print(CArray([[1,2],[2,4]], tosparse=True).norm_2d())
5.0
```

```
>>> print(CArray([[1,2],[2,4]]).norm_2d(axis=0).round(5))
CArray([[2.23607 4.47214]])
>>> print(CArray([[1,2],[2,4]]).norm_2d(axis=1).round(5))
CArray([[2.23607]
 [4.47214]])
```

```
>>> print(CArray([1,2,3]).norm_2d(2, axis=0))
CArray([[1. 2. 3.]])
>>> print(CArray([1,2,3]).norm_2d(2, axis=1).round(5))
CArray([[3.74166]])
```

```
>>> print(CArray([1,0,3], tosparse=True).norm_2d(axis=0))  # Norm is dense
CArray([[1. 0. 3.]])
>>> print(CArray([1,0,3], tosparse=True).norm_2d(axis=1).round(5))
CArray([[3.16228]])
```

normpdf (*self*, *mu=0.0*, *sigma=1.0*)

Return normal distribution function value with mean and standard deviation given for the current array values.

DENSE ARRAYS ONLY

The norm pdf is given by:

$$y = f(x|\mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{-(x-\mu)^2}{2\sigma^2}}$$

The standard normal distribution has $\mu = 0$ and $\sigma = 1$.

Parameters

mu [float, optional] Normal distribution mean. Default 0.0.

sigma [float, optional] Normal distribution standard deviation. Default 1.0.

Returns

CArray Normal distribution values.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([1,2,3]).normpdf())
CArray([0.241971 0.053991 0.004432])
```

```
>>> print(CArray([1,2,3]).normpdf(2,0.5))
CArray([0.107982 0.797885 0.107982])
```

classmethod ones (*shape*, *dtype=<class 'float'>*, *sparse=False*)

Return a new array of given shape and type, filled with ones.

Parameters

shape [int or tuple] Shape of the new array, e.g., 2 or (2,3).

dtype [str or dtype, optional] The desired data-type for the array. Default is float.

sparse [bool, optional] If False (default) a dense array will be returned. Otherwise, a sparse array of ones is created.

Returns

CArray Array of ones with the given properties.

Warning: When sparse is True, array is created as dense and then converted to sparse format. Consequently, the performance of this method is not comparable to other sparse array creation routines.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray.ones(2))
CArray([1. 1.])
```

```
>>> print(CArray.ones((2,1), dtype=int, sparse=True))
CArray(  (0, 0) 1
  (1, 0)        1)
```

pinv(*self*, *rcond=1e-15*)

Compute the (Moore-Penrose) pseudo-inverse of a matrix.

DENSE FORMAT ONLY

Calculate the generalized inverse of a matrix using its singular-value decomposition (SVD) and including all large singular values.

Parameters

rcond [float] Cutoff for small singular values. Singular values smaller (in modulus) than rcond * *largest_singular_value* (again, in modulus) are set to zero.

Returns

array_pinv [CArray] The pseudo-inverse of array. Resulting array have shape (array.shape[1], array.shape[0]).

Raises

LinAlgError [dense only] If array is not square or inversion fails.

Notes

The pseudo-inverse of a matrix A, denoted A^+, is defined as: "the matrix that 'solves' [the least-squares problem] $Ax = b$," i.e., if $\bar{x}$ is said solution, then A^+ is that matrix such that :math:'bar{x} = A^+b'. It can be shown that if $Q_1 \Sigma Q_2^T = A$ is the singular value decomposition of A, then $A^+ = Q_2 \Sigma^+ Q_1^T$, where $Q_{1,2}$ are orthogonal matrices, Σ is a diagonal matrix consisting of A's so-called singular values, (followed, typically, by zeros), and then Σ^+ is simply the diagonal matrix consisting of the reciprocals of A's singular values (again, followed by zeros). [1]

References

[1]

Examples

```
>>> from secml.array import CArray
```

The following example checks that: array * array_pinv * array == array and array_pinv * array * array_pinv == array_pinv

```
>>> array = CArray([[1,3],[0,5],[8,2]])
>>> array_pinv = array.pinv()
>>> (array == array.dot(array_pinv.dot(array)).round()).all()
True
```

```
>>> (array_pinv.round(2) == array_pinv.dot(array.dot(array_pinv)).round(2)).
↪all()
True
```

pow(*self*, *exp*)

Array elements raised to powers from input exponent, element-wise.

Raise each base in the array to the positionally-corresponding power in exp. exp must be broadcastable to the same shape of array. If exp is a scalar, works like standard ** operator.

Parameters

exp [CArray or scalar] Exponent of power, can be another array (DENSE ONLY) or a single scalar. If array, must have the same shape of original array.

Returns

CArray New array with the power of current data using input exponents.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([1,2,3]).pow(2))
CArray([1 4 9])
```

```
>>> print(CArray([1,2,3]).pow(CArray([2,0,3])))
CArray([ 1  1 27])
```

```
>>> print(CArray([1,0,3], tosparse=True).pow(2))
CArray(  (0, 0) 1
  (0, 2)        9)
```

prod(*self*, *axis=None*, *dtype=None*, *keepdims=True*)

Return the product of array elements over a given axis.

Parameters

axis [int or None, optional] Axis along which the product is computed. The default (None) is to compute the product over the flattened array.

dtype [str or dtype, optional] The data-type of the returned array, as well as of the accumulator in which the elements are multiplied. By default, if array is of integer type, dtype is the default platform integer. (Note: if the type of array is unsigned, then so is dtype.) Otherwise, the dtype is the same as that of array.

keepdims [bool, optional] If this is set to True (default), the result will broadcast correctly against the original array. Otherwise resulting array is flattened.

Returns

scalar or CArray Product of the elements in the array. If axis is None, scalar is returned. Otherwise, a CArray with shape and number of dimensions consistent with the original array and the axis parameter is returned.

Notes

Differently from numpy, we manage flat vectors as 2-Dimensional of shape (1, array.size). This means that when axis=0, a flat array is returned as is (see examples).

Arithmetic is modular when using integer types, and no error is raised on overflow. That means that, on a 32-bit platform:

```
>>> print(CArray([536870910, 536870910, 536870910, 536870910]).prod())  #␣
↪random result
16
```

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([[1,2],[3,4]]).prod())
24
```

```
>>> print(CArray([[1,2],[3,4]], tosparse=True).prod(axis=1))
CArray(  (0, 0) 2
  (1, 0)        12)
>>> print(CArray([[1,2],[3,4]]).prod(axis=0, dtype=float))
CArray([[3. 8.]])
```

```
>>> print(CArray([1,2,3]).prod(axis=0))
CArray([1 2 3])
>>> print(CArray([1,2,3]).prod(axis=1))
CArray([6])
```

classmethod rand(*shape*, *random_state=None*, *sparse=False*, *density=0.01*)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the “continuous uniform” distribution over the stated interval. To sample Unif[a, b), b > a multiply the output of rand by (b-a) and add a:

(b - a) * rand() + a

Parameters

shape [int or tuple] Shape of the new array.

random_state [int or None, optional] If int, random_state is the seed used by the random number generator; If None, is the seed used by np.random.

sparse [bool, optional] If False (default) a dense array will be returned. Otherwise, a sparse array of zeros is created.

density [scalar, optional, sparse only] Density of the generated sparse array, default 0.01 (1%). Density equal to one means a full array, density of 0 means no non-zero items.

Returns

CArray Array of random floats with the given shape and format.

Examples

```
>>> from secml.array import CArray
```

```
>>> array_dense = CArray.randn(shape=2)
>>> print(array_dense)
CArray([-0.170139  0.445385])
```

```
>>> array_dense = CArray.rand(shape=(2, 3))
>>> print(array_dense)
[[ 0.68588225  0.88371576  0.3958642 ]
 [ 0.58243871  0.05104796  0.77719998]]
```

```
>>> array_sparse = CArray.rand((2, 3), sparse=True, density=0.45)
>>> print(array_sparse)
CArray(  (0, 0) 0.209653887609
  (1, 1)        0.521906773406)
```

classmethod randint(*low*, *high=None*, *shape=None*, *random_state=None*, *sparse=False*)

Return random integers from low (inclusive) to high (exclusive).

Return random integers from the "discrete uniform" distribution in the "half-open" interval [low, high). If high is None (the default), then results are from [0, low).

Parameters

low [int] Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is the highest such integer).

high [int or None, optional] If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None).

shape [int, tuple or None, optional] Shape of output array. If None, a single value is returned.

random_state [int or None, optional] If int, random_state is the seed used by the random number generator; If None, is the seed used by np.random.

sparse [bool, optional] If False (default) a dense array will be returned. Otherwise, a random sparse array is created.

Returns

CArray Size-shaped array of random integers.

Warning: When sparse is True, array is created as dense and then converted to sparse format. Consequently, the performance of this method is not comparable to other sparse array creation routines.

Examples

```
>>> from secml.array import CArray
```

```
>>> array = CArray.randint(5, shape=10)
>>> print(array)
CArray([1 0 0 2 2 0 2 4 3 4])
```

```
>>> array = CArray.randint(0, 5, 10)
>>> print(array)
CArray([0 2 2 0 3 1 4 2 4 1])
```

```
>>> array = CArray.randint(0, 5, (2, 2))
>>> print(array)
CArray([[3 2]
 [0 2]])
```

classmethod randn(*shape*, *random_state=None*)

Return a sample (or samples) from the "standard normal" distribution.

DENSE FORMAT ONLY

The samples are generated from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1.

Parameters

shape [int or tuple] Shape of the new array.

random_state [int or None, optional] If int, random_state is the seed used by the random number generator; If None, is the seed used by np.random.

Returns

CArray or float A new array of given shape with floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

Examples

```
>>> from secml.array import CArray
```

```
>>> array_dense = CArray.randn(shape=2)
>>> print(array_dense)
CArray([-0.739091  1.201532])
```

```
>>> array_dense = CArray.randn(shape=(2, 3))
>>> print(array_dense)
CArray([[ 0.2848132  -0.02965108  1.41184901]
 [-1.3842878   0.2673215   0.18978747]])
```

classmethod randsample(*a*, *shape=None*, *replace=False*, *random_state=None*, *sparse=False*)

Generates a random sample from a given array.

Parameters

a [CArray or int] If an array, a random sample is generated from its elements. If an int, the random sample is generated as if a was CArray.arange(n)

shape [int, tuple or None, optional] Shape of output array. If None, a single value is returned.

replace [bool, optional] Whether the sample is with or without replacement, default False.

random_state [int or None, optional] If int, random_state is the seed used by the random number generator; If None, is the seed used by np.random.

sparse [bool, optional] If False (default) a dense array will be returned. Otherwise, a random sparse array is created.

Returns

CArray The generated random samples.

Warning: When sparse is True, array is created as dense and then converted to sparse format. Consequently, the performance of this method is not comparable to other sparse array creation routines.

Examples

```
>>> from secml.array import CArray
```

```
>>> array = CArray.randsample(10, shape=(2, 3))
>>> print(array)
CArray([[2 9 4]
 [8 6 5]])
```

```
>>> array = CArray.randsample(CArray([1,5,6,7,3]), shape=4)
>>> print(array)
CArray([3 7 5 6])
```

```
>>> CArray.randsample(3, 4)
Traceback (most recent call last):
    ...
ValueError: Cannot take a larger sample than population when 'replace=False'
```

classmethod randuniform(*low=0.0*, *high=1.0*, *shape=None*, *random_state=None*, *sparse=False*)

Return random samples from low (inclusive) to high (exclusive).

Samples are uniformly distributed over the half-open interval [low, high) (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn.

Parameters

low [float or CArray, optional]

Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.

A CArray of floats can be passed to specify a different bound for each position.

high [float or CArray, optional]

Upper boundary of the output interval. All values generated will be less than high. The default value is 1.0.

A CArray of floats can be passed to specify a different bound for each position.

shape [int, tuple or None, optional] Shape of output array. If None, a single value is returned.

random_state [int or None, optional]

If int, random_state is the seed used by the random number generator; If None, is the seed used by np.random.

sparse [bool, optional] If False (default) a dense array will be returned. Otherwise, a random sparse array is created.

Returns

CArray Size-shaped array of random samples.

Warning: When sparse is True, array is created as dense and then converted to sparse format. Consequently, the performance of this method is not comparable to other sparse array creation routines.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray.randuniform(high=5.0, shape=5))
CArray([ 4.36769   0.139844  3.711734  4.924484  3.737672])
```

```
>>> print(CArray.randuniform(shape=(2, 5)))
CArray([[ 0.158324  0.485235  0.723386  0.072326  0.344732]
 [ 0.761642  0.844458  0.501523  0.171417  0.002068]])
```

```
>>> print(CArray.randuniform(CArray([-1, -2, 3]), 5, (2, 3)))
CArray([[ -0.584032  1.433291  3.671319]
 [ 3.566163 -1.139602  4.268376]])
```

ravel (*self*)

Return a flattened array.

For dense format a 1-D array, containing the elements of the input, is returned. For sparse format a (1 x array.size) array will be returned.

A copy is made only if needed.

Returns

CArray Flattened view (if possible) of the array with shape (array.size,) for dense format or (1, array.size) for sparse format.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([[1,2],[3,4]]).ravel())
CArray([1 2 3 4])
```

```
>>> print(CArray([[1],[2],[3]], tosparse=True).ravel())
CArray(  (0, 0) 1
  (0, 1)        2
  (0, 2)        3)
```

repeat (*self*, *repeats*, *axis=None*)

Repeat elements of an array.

DENSE FORMAT ONLY

Parameters

repeats [int, list or CArray] The number of repetitions for each element. If this is an array_like object, will be broadcasted to fit the shape of the given axis.

axis [int, optional] The axis along which to repeat values. By default, array is flattened before use.

Returns

CArray Output array which has the same shape as original array, except along the given axis. If axis is None, a flat array is returned.

Examples

```
>>> from secml.array import CArray
```

```
>>> x = CArray([[1,2],[3,4]])
```

```
>>> print(x.repeat(2))
CArray([1 1 2 2 3 3 4 4])
```

```
>>> print(x.repeat(2, axis=1))  # Repeat the columns on the right
CArray([[1 1 2 2]
 [3 3 4 4]])
>>> print(x.repeat(2, axis=0))  # Repeat the rows on the right
CArray([[1 2]
 [1 2]
 [3 4]
 [3 4]])
```

```
>>> print(x.repeat([1, 2], axis=0))
CArray([[1 2]
 [3 4]
 [3 4]])
```

```
>>> x.repeat([1, 2])  # repeats size must be consistent with axis
Traceback (most recent call last):
    ...
ValueError: operands could not be broadcast together with shape (4,) (2,)
```

```
>>> x = CArray([1,2,3])
>>> print(x.repeat(2, axis=0))  # Repeat the (only) row on the right
CArray([1 1 2 2 3 3])
>>> print(x.repeat(2, axis=1))  # No columns to repeat
Traceback (most recent call last):
  ...
numpy.AxisError: axis 1 is out of bounds for array of dimension 1
```

repmat (*self*, *m*, *n*)

Repeat an array M x N times.

Parameters

m, n [int] The number of times the array is repeated along the first and second axes.

Returns

CArray The result of repeating array m X n times.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([[1,2]],tosparse=True).repmat(2,2))
CArray(  (0, 0) 1
  (0, 1)        2
  (0, 2)        1
  (0, 3)        2
  (1, 0)        1
  (1, 1)        2
  (1, 2)        1
  (1, 3)        2)
```

```
>>> print(CArray([1,2]).repmat(2,2))
CArray([[1 2 1 2]
 [1 2 1 2]])
>>> print(CArray([1,2]).repmat(1,2))
CArray([[1 2 1 2]])
>>> print(CArray([1,2]).repmat(2,1))
CArray([[1 2]
 [1 2]])
```

reshape (*self*, *newshape*)

Gives a new shape to an array without changing its data.

Parameters

newshape [int or sequence of ints] Desired shape for output array. If an integer or a tuple of length 1, resulting array will have shape (n,) if dense, (1,n) if sparse.

A copy is made only if needed.

Returns

CArray Array with new shape. If possible, a view of original array data will be returned, otherwise a copy will be made first.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([1,2,3]).reshape((3,1)))
CArray([[1]
 [2]
 [3]])
```

```
>>> print(CArray([[1],[2],[3]], tosparse=True).reshape(3))
CArray(  (0, 0) 1
  (0, 1)        2
  (0, 2)        3)
```

```
>>> CArray([1,2,3]).reshape(4)
Traceback (most recent call last):
  ...
ValueError: cannot reshape array of size 3 into shape (4,)
```

resize (*self*, *newshape*, *constant=0*)

Return a new array with the specified shape.

Missing entries are filled with input constant (default 0).

DENSE FORMAT ONLY

Parameters

newshape [int or sequence of ints] Integer or one integer for each desired dimension of output array. If a tuple of length 1, output sparse array will have shape (1, n).

constant [scalar] Scalar to be used for filling missing entries. Default 0.

Returns

CArray Array with new shape. Array dtype is preserved. Missing entries are filled with the desired constant (default 0).

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([1,2,3]).resize((3,3)))
CArray([[1 2 3]
 [0 0 0]
 [0 0 0]])
```

```
>>> print(CArray([1,2,3]).resize((3,1)))
CArray([[1]
 [2]
 [3]])
```

```
>>> print(CArray([1,2,3]).resize((1,3)))
CArray([[1 2 3]])
```

```
>>> print(CArray([[1,2,3]]).resize((5, )))
CArray([1 2 3 0 0])
```

```
>>> from secml.core.constants import inf
>>> print(CArray([[1,2,3]]).resize((5, ), constant=inf))
CArray([                   1                    2                    3
 -9223372036854775808 -9223372036854775808])
```

```
>>> print(CArray([[0, 1],[2, 3]]).resize(3))
CArray([0 1 2])
```

```
>>> print(CArray([[0, 1],[2, 3]]).resize((3, 3)))
CArray([[0 1 2]
 [3 0 0]
 [0 0 0]])
```

```
>>> print(CArray([[0, 1, 2],[3, 4, 5]]).resize((2, 2)))
CArray([[0 1]
 [2 3]])
```

round(*self*, *decimals=0*)

Evenly round to the given number of decimals.

Parameters

decimals [int, optional] Number of decimal places to round to (default: 0). If decimals is negative, it specifies the number of positions to round to the left of the decimal point.

Returns

out_rounded [CArray] An new array containing the rounded values. The real and imaginary parts of complex numbers are rounded separately. The result of rounding a float is a float.

See also:

ceil Return the ceiling of the input, element-wise.

floor Return the floor of the input, element-wise.

Notes

For values exactly halfway between rounded decimal values, we rounds to the nearest even value. Thus 1.5 and 2.5 round to 2.0, -0.5 and 0.5 round to 0.0, etc. Results may also be surprising due to the inexact representation of decimal fractions in the IEEE floating point standard [1] and errors introduced when scaling by powers of ten.

References

[1], [2]

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([1.28,5.62]).round())
CArray([1. 6.])
```

```
>>> print(CArray([1.28,5.62],tosparse=True).round(decimals=1))
CArray(  (0, 0) 1.3
  (0, 1)        5.6)
```

```
>>> print(CArray([.5, 1.5, 2.5, 3.5, 4.5]).round()) # rounds to nearest even␣
↪value
CArray([0. 2. 2. 4. 4.])
```

```
>>> print(CArray([1,5,6,11]).round(decimals=-1))
CArray([ 0  0 10 10])
```

save (*self*, *datafile*, *overwrite=False*)

Save array data into plain text file.

Data is stored preserving original data type.

The default encoding is *utf-8*.

Parameters

datafile [str, file_handle (dense only)] Text file to save data to. If a string, it's supposed to be the filename of file to save. If a file handle, data will be stored using active file handle mode. If the filename ends in .gz, the file is automatically saved in compressed gzip format. load() function understands gzipped files transparently.

overwrite [bool, optional] If True and target file already exists, file will be overwritten. Otherwise (default), IOError will be raised.

Notes

- Dense format, flat arrays are stored with shape N x 1.
- **Sparse format, we only save non-zero data along with indices** necessary to reconstruct original 2-dimensional array.
- **Dense format, shape of original array can be easily recognized** from target text file.

sha1 (*self*)

Calculate the sha1 hexadecimal hash of array.

Returns

hash [str] Hexadecimal hash of array.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([0,1,3]).sha1())
9d9d15176c022373488fb8a2b34be0ba3046f5c6
```

property shape
Shape of stored data, tuple of ints.

shuffle(*self*)
Modify array in-place by shuffling its contents.

This function only shuffles the array along the first index of a not vector-like, multi-dimensional array.

Examples

```
>>> from secml.array import CArray
```

```
>>> array = CArray([2,3,0,1])
>>> array.shuffle()
>>> print(array)
CArray([0 2 1 3])  # random result
```

```
>>> array = CArray([[2,3],[0,1]])
>>> array.shuffle()
>>> print(array)
CArray([[0 1]
 [2 3]])
```

sign(*self*)
Returns element-wise sign of the array.

The sign function returns -1 if x < 0, 0 if x == 0, 1 if x > 0.

Returns

CArray Array with sign of each element.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([[-2,0,2]]).sign())
CArray([[-1  0  1]])
```

```
>>> print(CArray([-2,0,2], tosparse=True).sign())
CArray(  (0, 0) -1
  (0, 2)        1)
```

sin(*self*)
Trigonometric sine, element-wise.

DENSE FORMAT ONLY

The array elements are considered angles, in radians (2π rad equals 360 degrees).

Returns

CArray New array with trigonometric sine element-wise.

Notes

The sine is one of the fundamental functions of trigonometry (the mathematical study of triangles). Consider a circle of radius 1 centered on the origin. A ray comes in from the $+x$ axis, makes an angle at the origin (measured counter-clockwise from that axis), and departs from the origin. The y coordinate of the outgoing ray's intersection with the unit circle is the sine of that angle. It ranges from -1 for $x = 3\pi/2$ to +1 for $\pi/2$. The function has zeroes where the angle is a multiple of π. Sines of angles between π and 2π are negative. The numerous properties of the sine and related functions are included in any standard trigonometry text.

Examples

```
>>> from secml.array import CArray
>>> from secml.core.constants import pi
```

```
>>> print((CArray([0,90,180,270,360,-90,-180,-270])*pi/180).sin().round())
CArray([ 0.  1.  0. -1. -0. -1. -0.  1.])
```

```
>>> print((CArray([[45,135],[225,315]])*pi/180).sin())
CArray([[ 0.707107  0.707107]
 [-0.707107 -0.707107]])
```

property size

Size (number of elements) of array.

For sparse data, this counts both zeros and non-zero elements.

sort(*self*, *axis=-1*, *kind='quicksort'*, *inplace=False*)

Sort an array.

Parameters

axis [int, optional] Axis along which to sort. The default is -1 (the last axis).

kind [{'quicksort', 'mergesort', 'heapsort'}, optional] Sorting algorithm to use. Default 'quicksort'. For sparse arrays, only 'quicksort' is available.

inplace [bool, optional] If True, array will be sorted in-place. Default False.

Returns

CArray Sorted array.

See also:

numpy.sort Description of different sorting algorithms.

CArray.argsort Indirect sort.

Notes

Differently from numpy, we manage flat vectors as 2-Dimensional of shape (1, array.size). This means that when axis=0, flat array is returned as is (see examples).

For large sparse arrays is actually faster to convert to dense first.

Examples

```
>>> from secml.array import CArray
```

```
>>> array = CArray([5,-1,0,-3])
>>> print(array.sort())
CArray([-3 -1  0  5])
```

```
>>> array = CArray([5,-1,0,-3])
>>> print(array.sort(axis=0))
CArray([ 5 -1  0 -3])
```

```
>>> array = CArray([5,-1,0,-3])
>>> print(array.sort(axis=1))
CArray([-3 -1  0  5])
```

```
>>> array = CArray([5,-1,0,-3])
>>> out = array.sort(inplace=True)
>>> print(out)
CArray([-3 -1  0  5])
>>> array[0] = 100
>>> print(out)
CArray([100  -1   0   5])
```

sqrt (*self*)

Compute the positive square-root of an array, element-wise.

If any array element is complex, a complex array is returned (and the square-roots of negative reals are calculated). If all of the array elements are real, so is the resulting array, with negative elements returning nan.

Returns

CArray A new array with the element-wise positive square-root of original array.

Notes

sqrt has, consistent with common convention, its branch cut the real "interval" *[-inf, 0)*, and is continuous from above on it. A branch cut is a curve in the complex plane across which a given complex function fails to be continuous.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray(2).sqrt())
CArray([1.414214])
```

```
>>> print(CArray([2,3,4]).sqrt())
CArray([1.414214 1.732051 2.      ])
```

```
>>> print(CArray([[2,3],[4,5]],tosparse=True).sqrt().round(4))
CArray(  (0, 0) 1.4142
  (0, 1)        1.7321
  (1, 0)        2.0
  (1, 1)        2.2361)
```

```
>>> print(CArray([-3, 0]).sqrt())
CArray([nan  0.])
```

std(*self*, *axis=None*, *ddof=0*, *keepdims=True*)

Compute the standard deviation along the specified axis.

Returns the standard deviation, a measure of the spread of a distribution, of the array elements. The standard deviation is computed for the flattened array by default, otherwise over the specified axis.

Parameters

axis [int, optional] Axis along which the standard deviation is computed. The default is to compute the standard deviation of the flattened array.

ddof [int, optional] Means Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements. By default ddof is zero.

keepdims [bool, optional] If this is set to True (default), the result will broadcast correctly against the original array.

Returns

float or CArray Standard deviation of the elements in the array. If axis is None, float is returned. Otherwise, a CArray with shape and number of dimensions consistent with the original array and the axis parameter is returned.

Notes

The standard deviation is the square root of the average of the squared deviations from the mean, i.e., 'std = sqrt(mean(abs(x - x.mean())**2))'.

The average squared deviation is normally calculated as 'x.sum() / N', where 'N = len(x)'. If, however, *ddof* is specified, the divisor 'N - ddof' is used instead. In standard statistical practice, 'ddof=1' provides an unbiased estimator of the variance of the infinite population. 'ddof=0' provides a maximum likelihood estimate of the variance for normally distributed variables. The standard deviation computed in this function is the square root of the estimated variance, so even with 'ddof=1', it will not be an unbiased estimate of the standard deviation per se.

Note that, for complex numbers, *std* takes the absolute value before squaring, so that the result is always real and not-negative.

For floating-point input, the mean is computed using default float precision. Depending on the input data, this can cause the results to be inaccurate, especially for 32-bit machines (float32).

Examples

```
>>> from secml.array import CArray
```

```
>>> print(round(CArray([[1,4],[4,3]],tosparse=True).std(), 2))
1.22
```

```
>>> print(CArray([[1,4],[4,3]],tosparse=True).std(axis=0))
CArray([[1.5 0.5]])
```

```
>>> print(CArray([[1,4],[4,3]]).std(axis=0, ddof=1).round(2))
CArray([[2.12 0.71]])
```

```
>>> print(CArray([1,4,4,3]).std(axis=0))
CArray([0. 0. 0. 0.])
>>> print(CArray([1,4,4,3]).std(axis=1).round(2))
CArray([1.22])
```

sum(*self*, *axis=None*, *keepdims=True*)

Sum of array elements over a given axis.

Parameters

axis [int or None, optional] Axis along which a sum is performed. The default (axis = None) is perform a sum over all the dimensions of the input array. axis may be negative, in which case it counts from the last to the first axis.

keepdims [bool, optional] If this is set to True (default), the result will broadcast correctly against the original array. Otherwise resulting array is flattened.

Returns

scalar or CArray Sum of the elements in the array. If axis is None, float is returned. Otherwise, a CArray with shape and number of dimensions consistent with the original array and the axis parameter is returned.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([-3,0,2]).sum())
-1
```

```
>>> print(CArray([[-3,0],[1,2]], tosparse=True).sum(axis=1))
CArray([[-3]
 [ 3]])
```

```
>>> print(CArray([-3,0,1,2]).sum(axis=0))
CArray([-3  0  1  2])
>>> print(CArray([-3,0,1,2]).sum(axis=1))
CArray([0])
```

t

tocoo(*self*)

Return a sparse scipy.sparse.coo_matrix representation of array.

Returns

scipy.sparse.coo_matrix A representation of current data as scipy.sparse.coo_matrix. If possible, we avoid copying original data.

Examples

```
>>> from secml.array import CArray
```

```
>>> array = CArray([[1,2],[0,4]], tosparse=True).tocoo()
>>> print(array)
  (0, 0)        1
  (0, 1)        2
  (1, 1)        4
>>> type(array)
<class 'scipy.sparse.coo.coo_matrix'>
```

```
>>> array = CArray([1,2,3]).tocoo()
>>> print(array)
  (0, 0)        1
  (0, 1)        2
  (0, 2)        3
>>> type(array)
<class 'scipy.sparse.coo.coo_matrix'>
```

tocsc(*self*)

Return a sparse scipy.sparse.csc_matrix representation of array.

Returns

scipy.sparse.csc_matrix A representation of current data as scipy.sparse.csc_matrix. If possible, we avoid copying original data.

Examples

```
>>> from secml.array import CArray
```

```
>>> array = CArray([[1,2],[0,4]], tosparse=True).tocsc()
>>> print(array)
  (0, 0)        1
  (0, 1)        2
  (1, 1)        4
>>> type(array)
<class 'scipy.sparse.csc.csc_matrix'>
```

```
>>> array = CArray([1,2,3]).tocsc()
>>> print(array)
  (0, 0)        1
  (0, 1)        2
  (0, 2)        3
```

(continues on next page)

(continued from previous page)

```
>>> type(array)
<class 'scipy.sparse.csc.csc_matrix'>
```

tocsr(*self*)

Return a sparse scipy.sparse.csr_matrix representation of array.

Returns

scipy.sparse.csr_matrix A representation of current data as scipy.sparse.csr_matrix. If possible, we avoid copying original data.

Examples

```
>>> from secml.array import CArray
```

```
>>> array = CArray([[1,2],[0,4]], tosparse=True).tocsr()
>>> print(array)
  (0, 0)        1
  (0, 1)        2
  (1, 1)        4
>>> type(array)
<class 'scipy.sparse.csr.csr_matrix'>
```

```
>>> array = CArray([1,2,3]).tocsr()
>>> print(array)
  (0, 0)        1
  (0, 1)        2
  (0, 2)        3
>>> type(array)
<class 'scipy.sparse.csr.csr_matrix'>
```

todense(*self*, *dtype=None*, *shape=None*)

Converts array to dense format.

Return current array if it has already a dense format.

Parameters

dtype [str or dtype, optional] Typecode or data-type to which the array is cast.

shape [sequence of ints, optional] Shape of the new array, e.g., '(2, 3)'.

Returns

CArray Dense array with input data and desired dtype and/or shape.

Notes

If current array has already a dense format, *dtype* and *shape* parameters will not be functional. Use *.astype()* or *.reshape()* function to alter array shape/dtype.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([[2, 0], [3, 4]], tosparse=True).todense(dtype=float))
CArray([[2. 0.]
 [3. 4.]])
```

```
>>> print(CArray([[2, 0], [3, 4]], tosparse=True).todense(shape=(4,)))
CArray([2 0 3 4])
```

todia (*self*)

Return a sparse scipy.sparse.dia_matrix representation of array.

Returns

scipy.sparse.dia_matrix A representation of current data as scipy.sparse.dia_matrix. If possible, we avoid copying original data.

Examples

```
>>> from secml.array import CArray
```

```
>>> array = CArray([[1,2],[0,4]], tosparse=True).todia()
>>> print(array)
  (0, 0)        1
  (1, 1)        4
  (0, 1)        2
>>> type(array)
<class 'scipy.sparse.dia.dia_matrix'>
```

```
>>> array = CArray([1,2,3]).todia()
>>> print(array)
  (0, 0)        1
  (0, 1)        2
  (0, 2)        3
>>> type(array)
<class 'scipy.sparse.dia.dia_matrix'>
```

todok (*self*)

Return a sparse scipy.sparse.dok_matrix representation of array.

Returns

scipy.sparse.dok_matrix A representation of current data as scipy.sparse.dok_matrix. If possible, we avoid copying original data.

Examples

```
>>> from secml.array import CArray
```

```
>>> array = CArray([[1,2],[0,4]], tosparse=True).todok()
>>> print(array)
  (0, 1)        2
  (0, 0)        1
  (1, 1)        4
>>> type(array)
<class 'scipy.sparse.dok.dok_matrix'>
```

```
>>> array = CArray([1,2,3]).todok()
>>> print(array)
  (0, 1)        2
  (0, 0)        1
  (0, 2)        3
>>> type(array)
<class 'scipy.sparse.dok.dok_matrix'>
```

tolil (*self*)

Return a sparse scipy.sparse.lil_matrix representation of array.

Returns

scipy.sparse.lil_matrix A representation of current data as scipy.sparse.lil_matrix. If possible, we avoid copying original data.

Examples

```
>>> from secml.array import CArray
```

```
>>> array = CArray([[1,2],[0,4]], tosparse=True).tolil()
>>> print(array)
  (0, 0)        1
  (0, 1)        2
  (1, 1)        4
>>> type(array)
<class 'scipy.sparse.lil.lil_matrix'>
```

```
>>> array = CArray([1,2,3]).tolil()
>>> print(array)
  (0, 0)        1
  (0, 1)        2
  (0, 2)        3
>>> type(array)
<class 'scipy.sparse.lil.lil_matrix'>
```

tolist (*self*)

Return the array as a (possibly nested) list.

Return a copy of the array data as a (nested) Python list. Data items are converted to the nearest compatible Python type.

Returns

list The possibly nested list of array elements.

Examples

```
>>> from secml.array import CArray
```

```
>>> array = CArray([[1,2],[0,4]]).tolist()
>>> array
[[1, 2], [0, 4]]
>>> print(CArray(array))
CArray([[1 2]
 [0 4]])
```

```
>>> print(CArray(array, tosparse=True))
CArray(  (0, 0) 1
  (0, 1)        2
  (1, 1)        4)
```

tondarray(*self*)

Return a dense numpy.ndarray representation of array.

Returns

numpy.ndarray A representation of current data as numpy.ndarray. If possible, we avoid copying original data.

Examples

```
>>> from secml.array import CArray
```

```
>>> array = CArray([1,2,3]).tondarray()
>>> array
array([1, 2, 3])
>>> type(array)
<class 'numpy.ndarray'>
```

```
>>> array = CArray([[1,2],[0,4]],tosparse=True).tondarray()
>>> array
array([[1, 2],
       [0, 4]], dtype=int64)
>>> type(array)
<class 'numpy.ndarray'>
```

tosparse(*self*, *dtype=None*, *shape=None*)

Converts array to sparse format.

Return current array if it has already a sparse format.

Parameters

dtype [str or dtype, optional] Typecode or data-type to which the array is cast.

shape [sequence of ints, optional] Shape of the new array, e.g., '(2, 3)'. Only 2-Dimensional sparse arrays are supported.

Returns

CArray Sparse array with input data and desired dtype and/or shape.

Notes

If current array has already a sparse format, *dtype* and *shape* parameters will not be functional. Use *.astype()* or *.reshape()* function to alter array shape/dtype.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([[2, 0], [3, 4]]).tosparse(dtype=float))
CArray(  (0, 0) 2.0
  (1, 0)        3.0
  (1, 1)        4.0)
```

```
>>> print(CArray([[2, 0], [3, 4]]).tosparse(shape=(1, 4)))
CArray(  (0, 0) 2
  (0, 2)        3
  (0, 3)        4)
```

transpose (*self*)

Returns current array with axes transposed.

A view is returned if possible.

Returns

CArray A view, if possible, of current array with axes suitably permuted.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([1, 2, 3]).transpose())
CArray([[1]
 [2]
 [3]])
```

```
>>> print(CArray([[1], [2], [3]]).transpose())
CArray([[1 2 3]])
```

unique (*self*, *return_index=False*, *return_inverse=False*, *return_counts=False*)

Find the unique elements of an array.

There are three optional outputs in addition to the unique elements: - the indices of the input array that give the unique values - the indices of the unique array that reconstruct the input array - the number of times each unique value comes up in the input array

Parameters

return_index [bool, optional] If True, also return the indices of array that result in the unique array (default False).

return_inverse [bool, optional, dense only] If True, also return the indices of the unique array that can be used to reconstruct the original array (default False).

return_counts [bool, optional] If True, also return the number of times each unique item appears.

Returns

unique [CArray] Dense array with the sorted unique values of the array.

unique_index [CArray, optional] The indices of the first occurrences of the unique values in the (flattened) original array. Only provided if return_index is True.

unique_counts [CArray, optional] The number of times each unique item appears in the original array. Only provided if return_counts is True.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray([[1,0,2],[2,0,3]]).unique())
CArray([0 1 2 3])
```

```
>>> print(CArray([1,2,2,3,3], tosparse=True).unique())
CArray([1 2 3])
```

```
>>> u, u_idx, u_inv = CArray([[2,2,3,3]]).unique(return_index=True, return_
↪inverse=True)
>>> print(u)  # unique
CArray([2 3])
>>> print(u_idx)  # unique_indices
CArray([0 2])
>>> print(u[u_inv])  # original (flattened) array reconstructed from unique_
↪inverse
CArray([2 2 3 3])
```

```
>>> u, u_counts = CArray([[2,2,2,3,3]]).unique(return_counts=True)
>>> print(u_counts)  # The number of times each unique item appears
CArray([3 2])
```

classmethod zeros(*shape*, *dtype=<class 'float'>*, *sparse=False*)

Return a new array of given shape and type, filled with zeros.

Parameters

shape [int or tuple] Shape of the new array, e.g., 2 or (2,3).

dtype [str or dtype, optional] The desired data-type for the array. Default is float.

sparse [bool, optional] If False (default) a dense array will be returned. Otherwise, a sparse array of zeros is created. Note that sparse arrays with only zeros appear empty when printing.

Returns

CArray Array of zeros with the given properties.

Examples

```
>>> from secml.array import CArray
```

```
>>> print(CArray.zeros(2))
CArray([0. 0.])
```

```
>>> array = CArray.zeros((2,1), dtype=int, sparse=True)
>>> print(array)  # sparse arrays with only zeros appear empty...
CArray()
>>> print(array.shape)
(2, 1)
```

array_utils

secml.array.array_utils.**is_vector_index**(*idx*)

Check if input index is valid for vector-like arrays.

An array is vector-like when 1-Dimensional or 2-Dimensional with shape[0] == 1.

Parameters

idx [int, bool, slice] Index to check.

Returns

out_check [bool] Return True if input is a valid index for any axis with size 1, else False.

secml.array.array_utils.**tuple_atomic_tolist**(*idx*)

Convert tuple atomic elements to list.

Atomic objects converted:

- *int*, *np.integer*
- *bool*, *np.bool_*

Parameters

idx [tuple] Tuple which elements have to be converted.

Returns

out_tuple [tuple] Converted tuple.

secml.array.array_utils.**tuple_sequence_tondarray**(*idx*)

Convert sequences inside tuple to ndarray.

A sequence can be:

- int, *np.integer*
- bool, *np.bool_*
- list
- *np.ndarray*
- CDense
- CSparse (are converted to dense first)

- CArray

Parameters

idx [tuple] Tuple which elements have to be converted.

Returns

out_tuple [tuple] Converted tuple.

4.7.6 secml.data

secml.data.loader

CDataLoader

class secml.data.loader.c_dataloader.**CDataLoader**

Bases: *secml.core.c_creator.CCreator*

Interface for Dataset loaders.

Attributes

class_type Defines class type.

logger Logger for current object.

verbose Verbosity level of logger output.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(self, *args, **kwargs)	Loads a dataset.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

abstract load(*self*, **args*, ***kwargs*)

Loads a dataset.

This method should return a *.CDataset* object.

CDataLoaderCIFAR

class secml.data.loader.c_dataloader_cifar.**CDataLoaderCIFAR**

Bases: *secml.data.loader.c_dataloader.CDataLoader*

Loads the CIFAR tiny images datasets.

Available at: https://www.cs.toronto.edu/~kriz/cifar.html

Attributes

class_type Defines class type.

data_md5 MD5 digest of the datafile.

data_path URL of the data directory.

data_url URL of the datafile.

logger Logger for current object.

verbose Verbosity level of logger output.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(self[, val_size])	Load all images of the dataset.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

abstract property data_md5

MD5 digest of the datafile. Specific for each dataset type.

Returns

str Expected MD5 digest of the dataset file.

abstract property data_path

URL of the data directory. Specific for each dataset type.

Returns

str Path to the folder where dataset data is stored.

abstract property data_url
URL of the datafile. Specific for each dataset type.

Returns

str URL of the remote datafile with dataset data.

abstract load(*self*, *val_size=0*)
Load all images of the dataset.

Each image is flattened. The first 1024 entries contain the red channel values, the next 1024 the green, and the final 1024 the blue. The image is stored in row-major order, so that the first 32 entries of the array are the red channel values of the first row of the image. Dtype of images is *uint8*. Dtype of labels is *int32*.

Extra dataset attributes:

- 'img_w', 'img_h': size of the images in pixels.
- 'class_names': dictionary with the original name of each class.

Parameters

val_size [int, optional] Size of the validation set. Default 0, so no validation dataset will be returned.

Returns

training_set [CDataset] Training set.

test_set [CDataset] Test set.

validation_set [CDataset, optional] Validation set. Returned only if val_size > 0.

class secml.data.loader.c_dataloader_cifar.**CDataLoaderCIFAR10**
Bases: *secml.data.loader.c_dataloader_cifar.CDataLoaderCIFAR*

Loads the CIFAR-10 tiny images dataset.

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

Available at: https://www.cs.toronto.edu/~kriz/cifar.html

Attributes

class_type ['CIFAR-10'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.

Continued on next page

Table 6 – continued from previous page

load(self[, val_size])	Load all images of the dataset.Load all images of the dataset.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property data_md5
MD5 digest of the datafile.

Returns

str Expected MD5 digest of the dataset file.

property data_path
URL of the data directory.

Returns

str Path to the folder where dataset data is stored.

property data_url
URL of the remote datafile.

Returns

str URL of the remote datafile with dataset data.

load(*self*, *val_size=0*)
Load all images of the dataset.Load all images of the dataset.

Each image is flattened. The first 1024 entries contain the red channel values, the next 1024 the green, and the final 1024 the blue. The image is stored in row-major order, so that the first 32 entries of the array are the red channel values of the first row of the image. Dtype of images is *uint8*. Dtype of labels is *int32*.

Extra dataset attributes:

- 'img_w', 'img_h': size of the images in pixels.
- 'class_names': dictionary with the original name of each class.

Parameters

val_size [int, optional] Size of the validation set. Default 0, so no validation dataset will be returned.

Returns

training_set [CDataset] Training set.

test_set [CDataset] Test set.

validation_set [CDataset, optional] Validation set. Returned only if val_size > 0.

class secml.data.loader.c_dataloader_cifar.**CDataLoaderCIFAR100**
Bases: *secml.data.loader.c_dataloader_cifar.CDataLoaderCIFAR*

Loads the CIFAR-100 tiny images dataset.

The CIFAR-100 dataset consists of 60000 32x32 colour images in 100 classes, containing 600 images each. There are 500 training images and 100 testing images per class. The 100 classes in the CIFAR-100 are grouped into 20 superclasses. Each image comes with a "fine" label (the class to which it belongs) and a "coarse" label (the superclass to which it belongs).

Available at: https://www.cs.toronto.edu/~kriz/cifar.html

Attributes

class_type ['CIFAR-100'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(self[, val_size])	Load all images of the dataset.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property data_md5
MD5 digest of the datafile.

Returns

str Expected MD5 digest of the dataset file.

property data_path
URL of the data directory.

Returns

str Path to the folder where dataset data is stored.

property data_url
URL of the remote datafile.

Returns

str URL of the remote datafile with dataset data.

load(*self*, *val_size=0*)
Load all images of the dataset.

Each image is flattened. The first 1024 entries contain the red channel values, the next 1024 the green, and the final 1024 the blue. The image is stored in row-major order, so that the first 32 entries of the array are the red channel values of the first row of the image. Dtype of images is *uint8*. Dtype of labels is *int32*.

Extra dataset attributes:

- 'img_w', 'img_h': size of the images in pixels.
- 'class_names': dictionary with the original name of each class.

Parameters

val_size [int, optional] Size of the validation set. Default 0, so no validation dataset will be returned.

Returns

training_set [CDataset] Training set.

test_set [CDataset] Test set.

validation_set [CDataset, optional] Validation set. Returned only if val_size > 0.

CDataLoaderICubWorld

class secml.data.loader.c_dataloader_icubworld.**CDataLoaderICubWorld**

Bases: *secml.data.loader.c_dataloader.CDataLoader*

Interface for loaders of iCubWorld datasets.

iCubWorld is a set of computer vision datasets for robotic applications, developed by Istituto Italiano di Tecnologia (IIT), Genova, Italy.

REF: https://robotology.github.io/iCubWorld

Attributes

class_type Defines class type.

logger Logger for current object.

verbose Verbosity level of logger output.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(self, *args, **kwargs)	Loads a dataset.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.

Continued on next page

Table 8 – continued from previous page

save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

abstract load(*self*, **args*, ***kwargs*)
Loads a dataset.

This method should return a *.CDataset* object.

class secml.data.loader.c_dataloader_icubworld.**CDataLoaderICubWorld28**
Bases: *secml.data.loader.c_dataloader_icubworld.CDataLoaderICubWorld*

Loader for iCubWorld28 dataset.

The dataset consists in 28 objects divided in 7 categories, where each category includes 4 objects. For each object there are 4 different acquisition days for training and 4 for testing, with ~150 frames per acquisition.

Attributes

class_type ['icubworld28'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(self, ds_type[, day, icub7, ...])	Load the dataset.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

load(*self*, *ds_type*, *day='day4'*, *icub7=False*, *resize_shape=(128, 128)*, *crop_shape=None*, *normalize=True*)
Load the dataset.

The pre-cropped version of the images is loaded, with size 128 x 128. An additional resize/crop shape could be passed as input if needed.

Extra dataset attributes:

- 'img_w', 'img_h': size of the images in pixels.
- 'y_orig': CArray with the original labels of the objects.

Parameters

ds_type [str] Identifier of the dataset to download, either 'train' or 'test'.

day [str, optional] Acquisition day from which to load the images. Default 'day4'. The available options are: 'day1', 'day2', 'day3', 'day4'.

icub7 [bool or int, optional] If True, load a reduced dataset with 7 objects by taking the 3rd object for each category. Default False. If int, the Nth object for each category will be loaded.

resize_shape [tuple, optional] Images will be resized to (height, width) shape. Default (128, 128).

crop_shape [tuple or None, optional] If a tuple, a crop of (height, width) shape will be extracted from the center of each image. Default None.

normalize [bool, optional] If True, images are normalized between 0-1. Default True.

Returns

CDataset Output dataset.

CDataLoaderImgClients

class secml.data.loader.c_dataloader_imgclients.**CDataLoaderImgClients**

Bases: *secml.data.loader.c_dataloader.CDataLoader*

Loads a dataset of images and corresponding labels from 'clients.txt'.

Attributes

class_type ['img-clients'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(self, ds_path, img_format[, ...])	Load all images of specified format inside given path.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.

Continued on next page

Table 10 – continued from previous page

set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

load(*self*, *ds_path*, *img_format*, *label_dtype=None*, *load_data=True*)

Load all images of specified format inside given path.

Extra dataset attributes:

- 'id': last *ds_path* folder.
- 'img_w', 'img_h': size of the images in pixels.
- 'img_c': images number of channels.
- Any other custom attribute is retrieved from 'attributes.txt' file. Only attributes of *str* type are currently supported.

Parameters

ds_path [str] Full path to dataset folder.

img_format [str] Format of the files to load.

label_dtype [str or dtype, optional] Datatype of the labels. If None, labels will be strings.

load_data [bool, optional] If True (default) features will be stored. Otherwise store the paths to the files with dtype=object.

CDataLoaderImgFolders

class secml.data.loader.c_dataloader_imgfolders.**CDataLoaderImgFolders**

Bases: *secml.data.loader.c_dataloader.CDataLoader*

Loads a dataset of images where clients are specified as different folders.

Attributes

class_type ['img-folders'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(self, ds_path, img_format[, label_re, ...])	Load all images of specified format inside given path.

Continued on next page

Table 11 – continued from previous page

load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

load(*self*, *ds_path*, *img_format*, *label_re=None*, *label_dtype=None*, *load_data=True*)

Load all images of specified format inside given path.

The following custom CDataset attributes are available:

- 'id': last *ds_path* folder.
- 'img_w', 'img_h': size of the images in pixels.
- 'img_c': images number of channels.
- Any other custom attribute is retrieved from 'attributes.txt' file. Only attributes of *str* type are currently supported.

Any other custom attribute is retrieved from 'attributes.txt' file.

Parameters

ds_path [str] Full path to dataset folder.

img_format [str] Format of the files to load.

label_re [re, optional] Regular expression that identify the correct label. If None, the whole name of the leaf folder will be used as label.

label_dtype [str or dtype, optional] Datatype of the labels. If None, labels will be strings.

load_data [bool, optional] If True (default) features will be stored. Otherwise store the paths to the files with dtype=object.

CDataLoaderLFW

class secml.data.loader.c_dataloader_lfw.**CDataLoaderLFW**

Bases: *secml.data.loader.c_dataloader.CDataLoader*

Loads the LFW Labeled Faces in the Wild dataset.

This dataset is a collection of JPEG pictures of famous people collected on the internet, all details are available on the official website:

http://vis-www.cs.umass.edu/lfw/

Each picture is centered on a single face. Each pixel of each channel (color in RGB) is encoded by a float in range 0.0 - 1.0.

The task is called Face Recognition (or Identification): given the picture of a face, find the name of the person given a training set (gallery).

This implementation uses *sklearn.datasets.fetch_lfw_people* module.

Attributes

class_type ['lfw'] Defines class type.

Methods

clean_tmp()	Cleans temporary files created by the DB loader.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(self[, min_faces_per_person, funneled, ...])	Load LFW dataset.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

static clean_tmp()

Cleans temporary files created by the DB loader.

This method deletes the joblib-related files created while loading the database.

Does not delete the downloaded database archive.

load(*self*, *min_faces_per_person=None*, *funneled=True*, *color=False*)

Load LFW dataset.

Extra dataset attributes:

- 'img_w', 'img_h': size of the images in pixels.
- 'y_names': tuple with the name string for each class.

Parameters

min_faces_per_person [int or None, optional] The extracted dataset will only retain pictures of people that have at least min_faces_per_person different pictures. Default None, so all db images are returned.

funneled [bool, optional] Download and use the images aligned with deep funneling. Default True.

color [bool, optional] Keep the 3 RGB channels instead of averaging them to a single gray level channel. Default False.

CDataLoaderMNIST

class secml.data.loader.c_dataloader_mnist.**CDataLoaderMNIST**

Bases: *secml.data.loader.c_dataloader.CDataLoader*

Loads the MNIST Handwritten Digits dataset.

This dataset has a training set of 60,000 examples, and a test set of 10,000 examples. All images are 28 x 28 black and white 8bit (0 - 255).

Available at: http://yann.lecun.com/exdb/mnist/

Attributes

class_type ['mnist'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(self, ds[, digits, num_samples])	Load all images of specified format inside given path.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

load(*self*, *ds*, *digits=(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)*, *num_samples=None*)

Load all images of specified format inside given path.

Adapted from: http://cvxopt.org/_downloads/mnist.py

Extra dataset attributes:

- 'img_w', 'img_h': size of the images in pixels.
- 'y_original': array with the original labels (before renumbering)

Parameters

ds [str] Identifier of the dataset to download, either 'training' or 'testing'.

digits [tuple] Tuple with the digits to load. By default all digits are loaded.

num_samples [int or None, optional] Number of expected samples in resulting ds. If int, an equal number of samples will be taken from each class until *num_samples* have been loaded. If None, all samples will be loaded.

CDataLoaderPyTorch

class secml.data.loader.c_dataloader_pytorch.**CDataLoaderPyTorch**(*data*, *labels=None*, *batch_size=4*, *shuffle=False*, *transform=None*, *num_workers=0*)

Bases: object

Methods

get_loader	

get_loader(*self*)

CDataLoaderSkLearn

class secml.data.loader.c_dataloader_sklearn.**CDLRandom**(*n_samples=100*, *n_features=20*, *n_informative=2*, *n_redundant=2*, *n_repeated=0*, *n_classes=2*, *n_clusters_per_class=2*, *weights=None*, *flip_y=0.01*, *class_sep=1.0*, *hypercube=True*, *shift=0.0*, *scale=1.0*, *random_state=None*)

Bases: *secml.data.loader.c_dataloader.CDataLoader*

Class for loading random data.

Generate a random n-class classification problem.

This initially creates clusters of points normally distributed (std=1) about vertices of a 2 * class_sep-sided hypercube, and assigns an equal number of clusters to each class.

It introduces interdependence between these features and adds various types of further noise to the data.

Prior to shuffling, X stacks a number of these primary "informative" features, "redundant" linear combinations of these, "repeated" duplicates of sampled features, and arbitrary noise for and remaining features.

Parameters

n_samples [int, optional (default=100)] The number of samples.

n_features [int, optional (default=20)] The total number of features. These comprise n_informative informative features, n_redundant redundant features, n_repeated

duplicated features and `n_features - n_informative - n_redundant - n_repeated` useless features drawn at random.

n_informative [int, optional (default=2)] The number of informative features. Each class is composed of a number of gaussian clusters each located around the vertices of a hypercube in a subspace of dimension n_informative. For each cluster, informative features are drawn independently from N(0, 1) and then randomly linearly combined within each cluster in order to add covariance. The clusters are then placed on the vertices of the hypercube.

n_redundant [int, optional (default=2)] The number of redundant features. These features are generated as random linear combinations of the informative features.

n_repeated [int, optional (default=0)] The number of duplicated features, drawn randomly from the informative and the redundant features.

n_classes [int, optional (default=2)] The number of classes (or labels) of the classification problem.

n_clusters_per_class [int, optional (default=2)] The number of clusters per class.

weights [list of floats or None (default=None)] The proportions of samples assigned to each class. If None, then classes are balanced. Note that if `len(weights) == n_classes - 1`, then the last class weight is automatically inferred. More than n_samples samples may be returned if the sum of weights exceeds 1.

flip_y [float, optional (default=0.01)] The fraction of samples whose class are randomly exchanged.

class_sep [float, optional (default=1.0)] The factor multiplying the hypercube dimension.

hypercube [bool, optional (default=True)] If True, the clusters are put on the vertices of a hypercube. If False, the clusters are put on the vertices of a random polytope.

shift [float, array of shape [n_features] or None, optional (default=0.0)] Shift features by the specified value. If None, then features are shifted by a random value drawn in [-class_sep, class_sep].

scale [float, array of shape [n_features] or None, optional (default=1.0)] Multiply features by the specified value. If None, then features are scaled by a random value drawn in [1, 100]. Note that scaling happens after shifting.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, is the RandomState instance used by np.random.

Attributes

`class_type` ['classification'] Defines class type.

Methods

`copy`(self)	Returns a shallow copy of current class.
`create`([class_item])	This method creates an instance of a class with given type.
`deepcopy`(self)	Returns a deep copy of current class.
`get_class_from_type`(class_type)	Return the class associated with input type.
`get_params`(self)	Returns the dictionary of class hyperparameters.

Continued on next page

Table 14 – continued from previous page

get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(self)	Loads the dataset.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

load(*self*)

Loads the dataset.

Returns

dataset [CDataset] The randomly generated dataset.

class secml.data.loader.c_dataloader_sklearn.**CDLRandomRegression**(*n_samples=100*, *n_features=100*, *n_informative=10*, *n_targets=1*, *bias=0.0*, *effective_rank=None*, *tail_strength=0.5*, *noise=0.0*, *random_state=None*)

Bases: *secml.data.loader.c_dataloader.CDataLoader*

Generate a random regression problem.

The input set can either be well conditioned (by default) or have a low rank-fat tail singular profile.

The output is generated by applying a (potentially biased) random linear regression model with *n_informative* nonzero regressors to the previously generated input and some gaussian centered noise with some adjustable scale.

Parameters

n_samples [int, optional (default=100)] The number of samples.

n_features [int, optional (default=100)] The number of features.

n_informative [int, optional (default=10)] The number of informative features, i.e., the number of features used to build the linear model used to generate the output.

n_targets [int, optional (default=1)] The number of regression targets, i.e., the dimension of the y output vector associated with a sample. By default, the output is a scalar.

bias [float, optional (default=0.0)] The bias term in the underlying linear model.

effective_rank [int or None, optional (default=None)]

if not None: The approximate number of singular vectors required to explain most of the input data by linear combinations. Using this kind ofsingular spectrum in the input allows the generator to reproduce the correlations often observed in practice.

if None: The input set is well conditioned, centered and gaussian with unit variance.

tail_strength [float between 0.0 and 1.0, optional (default=0.5)] The relative importance of the fat noisy tail of the singular values profile if *effective_rank* is not None.

noise [float, optional (default=0.0)] The standard deviation of the gaussian noise applied to the output.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, is the RandomState instance used by np.random.

Attributes

class_type ['regression'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(self)	Loads the dataset.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

load(*self*)

Loads the dataset.

Returns

dataset [CDataset] The randomly generated dataset.

class secml.data.loader.c_dataloader_sklearn.**CDLRandomBlobs**(*n_samples=100*, *n_features=2*, *centers=3*, *cluster_std=1.0*, *center_box=(-10.0, 10.0)*, *random_state=None*)

Bases: *secml.data.loader.c_dataloader.CDataLoader*

Generate isotropic Gaussian blobs for clustering.

Parameters

n_samples [int, optional (default=100)] The total number of points equally divided among clusters.

n_features [int, optional (default=2)] The number of features for each sample. This parameter will not be considered if centers is different from None

centers [int or array of shape [n_centers, n_features]] The number of centers to generate (default=3), or the fixed center locations as list of tuples

cluster_std: float or sequence of floats, optional (default=1.0) The standard deviation of the clusters.

center_box [pair of floats (min, max), optional (default=(-10.0, 10.0))] The bounding box for each cluster center when centers are generated at random.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, is the RandomState instance used by np.random.

Attributes

class_type ['blobs'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(self)	Loads the dataset.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.

Continued on next page

Table 16 – continued from previous page

set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

load(*self*)
Loads the dataset.

Returns

dataset [CDataset] The randomly generated dataset.

class secml.data.loader.c_dataloader_sklearn.**CDLRandomBlobsRegression**(*n_samples=100, cluster_std=(1.0, 1.0), bias=1.0, w=(2.0, -1.0), centers=([0, 0], [-1, -1]), random_state=None*)

Bases: *secml.data.loader.c_dataloader.CDataLoader*

This class loads blobs regression.

Parameters

n_samples [int, optional (default=100)] The total number of points equally divided among clusters.

centers [int or array of shape [n_centers, n_features], optional (default=3)] The number of centers to generate, or the fixed center locations.

cluster_std: list of floats, optional (default=(1.0,1.0)) The standard deviation of the clusters.

bias [bias that will sum to the function]

w [the height of every gaussian]

centers: list of tuple optional (default=([1,1],[-1,-1])) The bounding box for each cluster center when centers are generated at random.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, is the RandomState instance used by np.random.

Attributes

class_type ['blobs-regression'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(self)	Loads the dataset.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

load(*self*)

Loads the dataset.

Returns

dataset [CDataset] The randomly generated dataset.

class secml.data.loader.c_dataloader_sklearn.**CDLRandomCircles**(*n_samples=100*, *noise=None*, *factor=0.8*, *random_state=None*)

Bases: *secml.data.loader.c_dataloader.CDataLoader*

Make a large circle containing a smaller circle in 2d.

Parameters

n_samples [int, optional (default=100)] The total number of points generated.

noise [double or None (default=None)] Standard deviation of Gaussian noise added to the data.

factor [double < 1 (default=.8)] Scale factor between inner and outer circle.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, is the RandomState instance used by np.random.

Attributes

class_type ['circles'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(self)	Loads the dataset.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

load(*self*)

Loads the dataset.

Returns

dataset [CDataset] The randomly generated dataset.

class secml.data.loader.c_dataloader_sklearn.**CDLRandomCircleRegression**(*n_samples=100*, *noise=None*, *factor=0.8*, *random_state=None*)

Bases: *secml.data.loader.c_dataloader.CDataLoader*

Make a large circle containing a smaller circle in 2d.

Parameters

n_samples [int, optional (default=100)] The total number of points generated.

noise [double or None (default=None)] Standard deviation of Gaussian noise added to the data.

factor [double < 1 (default=.8)] Scale factor between inner and outer circle.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, is the RandomState instance used by np.random.

Attributes

class_type ['circles-regression'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(self)	Loads the dataset.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

load(*self*)

Loads the dataset.

Returns

dataset [CDataset] The randomly generated dataset.

class secml.data.loader.c_dataloader_sklearn.**CDLRandomMoons**(*n_samples=100*, *noise=None*, *random_state=None*)

Bases: *secml.data.loader.c_dataloader.CDataLoader*

Make two interleaving half circles.

Parameters

n_samples [int, optional (default=100)] The total number of points generated.

noise [double or None (default=None)] Standard deviation of Gaussian noise added to the data.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, is the RandomState instance used by np.random.

Attributes

class_type ['moons'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(self)	Loads the dataset.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

load(*self*)

Loads the dataset.

Returns

dataset [CDataset] The randomly generated dataset.

class secml.data.loader.c_dataloader_sklearn.**CDLRandomBinary**(*n_samples=100*, *n_features=2*)

Bases: *secml.data.loader.c_dataloader.CDataLoader*

Generate random binary data.

Parameters

n_samples [int, optional (default=100)] The total number of points generated.

n_features [int, optional (default=2)] The total number of features

Attributes

class_type ['binary'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.

Continued on next page

Table 21 – continued from previous page

get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(self)	Loads the dataset.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

load(*self*)

Loads the dataset.

Returns

dataset [CDataset] The randomly generated dataset.

class secml.data.loader.c_dataloader_sklearn.**CDLIris**(*class_list=None*, *zero_one=False*)

Bases: secml.data.loader.c_dataloader_sklearn.CDLRandomToy

Loads Iris dataset.

The iris dataset is a classic and very easy multi-class classification dataset.

Classes	3
Samples per class	50
Samples total	150
Dimensionality	4
Features	real, positive

Parameters

class_list [list of str (default None)] Each string is the name of data's class that we want in the new dataset. If None every class will be keep

zero_one [bool] If is true, and class list is equal to two, will be assigned 0 at the label with lower value, 1 to the other.

Attributes

class_type ['iris'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(self)	Loads the dataset.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

toy = 'iris'

class secml.data.loader.c_dataloader_sklearn.**CDLDigits**(*class_list=None*, *zero_one=False*)

Bases: secml.data.loader.c_dataloader_sklearn.CDLRandomToy

Loads Digits dataset.

The digits dataset is a classic and very easy multi-class classification dataset. Each datapoint is a 8x8 image of a digit.

Classes	10
Samples per class	~180
Samples total	1797
Dimensionality	64
Features	integers 0-16

Parameters

class_list [list of str (default None)] Each string is the name of data's class that we want in the new dataset. If None every class will be keep

zero_one [bool] If is true, and class list is equal to two, will be assigned 0 at the label with lower value, 1 to the other.

Attributes

class_type ['digits'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(self)	Loads the dataset.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

toy = 'digits'

class secml.data.loader.c_dataloader_sklearn.**CDLBoston**(*class_list=None*, *zero_one=False*)

Bases: secml.data.loader.c_dataloader_sklearn.CDLRandomToy

Loads Boston dataset.

Boston house-prices dataset, useful for regression.

Samples total	506
Dimensionality	13
Features	real, positive
Targets	real 5. - 50.

Parameters

class_list [list of str (default None)] Each string is the name of data's class that we want in the new dataset. If None every class will be keep

zero_one [bool] If is true, and class list is equal to two, will be assigned 0 at the label with lower value, 1 to the other.

Attributes

class_type ['boston'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(self)	Loads the dataset.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

toy = 'boston'

class secml.data.loader.c_dataloader_sklearn.**CDLDiabetes**(*class_list=None*, *zero_one=False*)

Bases: secml.data.loader.c_dataloader_sklearn.CDLRandomToy

Loads Diabetes dataset.

Diabetes dataset, useful for regression.

Samples total	442
Dimensionality	10
Features	real, -.2 < x < .2
Targets	integer 25 - 346

Parameters

class_list [list of str (default None)] Each string is the name of data's class that we want in the new dataset. If None every class will be keep

zero_one [bool] If is true, and class list is equal to two, will be assigned 0 at the label with lower value, 1 to the other.

Attributes

class_type ['diabetes'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(self)	Loads the dataset.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

toy = 'diabetes'

CDataLoaderSvmLight

class secml.data.loader.c_dataloader_svmlight.**CDataLoaderSvmLight**

Bases: *secml.data.loader.c_dataloader.CDataLoader*

Loads and Saves data in svmlight / libsvm format.

Attributes

class_type ['svmlight'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
dump(d, f[, zero_based, comment])	Dumps a dataset in the svmlight / libsvm file format.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(self, file_path[, dtype_samples, ...])	Loads a dataset from the svmlight / libsvm format and returns a sparse dataset.

Continued on next page

Table 26 – continued from previous page

load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

static dump(*d*, *f*, *zero_based=True*, *comment=None*)

Dumps a dataset in the svmlight / libsvm file format.

This format is a text-based format, with one sample per line. It does not store zero valued features hence is suitable for sparse dataset.

The first element of each line can be used to store a target variable to predict.

Parameters

d [CDataset] Contain dataset with patterns and labels that we want store.

f [String] Path to file were we want store dataset into format svmlight or libsvm.

zero_based [bool, optional] Whether column indices should be written zero-based (True, default) or one-based (False).

comment [string, optional] Comment to insert at the top of the file. This should be either a Unicode string, which will be encoded as UTF-8, or an ASCII byte string. If a comment is given, then it will be preceded by one that identifies the file as having been dumped by scikit-learn. Note that not all tools grok comments in SVMlight files.

Examples

```
>>> from secml.data.loader import CDataLoaderSvmLight
>>> from secml.array import CArray
>>> patterns = CArray([[1,0,2], [4,0,5]])
>>> labels = CArray([0,1])
>>> CDataLoaderSvmLight.dump(CDataset(patterns,labels), "myfile.libsvm")
```

load(*self*, *file_path*, *dtype_samples=<class 'float'>*, *dtype_labels=<class 'float'>*, *n_features=None*, *zero_based=True*, *remove_all_zero=False*, *multilabel=False*, *load_infos=False*)

Loads a dataset from the svmlight / libsvm format and returns a sparse dataset.

Datasets must have only numerical feature indices and for every pattern indices must be ordered.

Extra dataset attributes:

- 'infos', CArray with inline comment for each sample.

Parameters

file_path [String] Path to file were dataset are stored into format svmlight or libsvm.

dtype_samples [str or dtype, optional] Data-type to which the samples should be casted. Default is float.

dtype_labels [str or dtype, optional] Data-type to which the labels should be casted. Default is float.

n_features [None or int, optional] The number of features to use. If None (default), it will be inferred. This argument is useful to load several files that are subsets of a bigger sliced dataset: each subset might not have examples of every feature, hence the inferred shape might vary from one slice to another.

zero_based: bool, optional Whether column indices are zero-based (True, default) or one-based (False). If column indices are set to be one-based, they are transformed to zero-based to match Python/NumPy conventions.

remove_all_zero: boolean, optional, default True If True every feature which is zero for every pattern will be removed from dataset.

multilabel [boolean, optional] True if every sample can have more than one label. Default False. (see http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multilabel.html)

load_infos [bool, optional] If True, inline comments will be loaded from the svmlight file and stored in the infos CDataset parameter (as CArray). Default False.

Returns

dataset [CDataset] Dataset object that contain patterns and labels. If *remove_all_zero* is set to True, the returned dataset will have the new argument *idx_mapping* with the mapping of the returned features to the original features's indices.

Examples

```
>>> from secml.data.loader import CDataLoaderSvmLight
>>> from secml.array import CArray
>>> patterns = CArray ([[1,0,2], [4,0,5]])
>>> labels = CArray ([0, 1])
>>> CDataLoaderSvmLight().dump(CDataset(patterns,labels), "myfile.libsvm")
>>> new_dataset = CDataLoaderSvmLight().load("myfile.libsvm", remove_all_
↪zero=True)
>>> print(new_dataset.X)
CArray(  (0, 1) 2.0
  (0, 0)        1.0
  (1, 1)        5.0
  (1, 0)        4.0)
>>> print(new_dataset.Y)
CArray([0. 1.])
>>> print(new_dataset.header.idx_mapping)
CArray([0 2])
```

CDataLoaderTorchDataset

class secml.data.loader.c_dataloader_torchvision.**CDataLoaderTorchDataset**(*tv_dataset_class*, ***kwargs*)

Bases: *secml.data.loader.c_dataloader.CDataLoader*

Wrapper for loading Torchvision datasets as CDatasets.

Parameters

tv_dataset_class [torch.Dataset] torchvision dataset class to load

Attributes

class_to_idx Dictionary for matching indexes and class names

class_type Defines class type.

logger Logger for current object.

verbose Verbosity level of logger output.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(self, *args, **kwargs)	Loads a dataset.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property class_to_idx
Dictionary for matching indexes and class names

load(*self*, **args*, ***kwargs*)
Loads a dataset.

This method should return a *.CDataset* object.

loader_utils

secml.data.loader.loader_utils.**resize_img**(*img*, *shape*)
Resize input image to desired shape.

If the input image is bigger than desired, the LANCZOS filter will be used. If calculates the output pixel value using a truncated sinc filter on all pixels that may contribute to the output value.

Otherwise, a LINEAR filter will be used. It calculates the output pixel value using linear interpolation on all pixels that may contribute to the output value.

Parameters

img [PIL.Image.Image] Image to be resized.

shape [tuple] Desired output image dimensions (height, width).

Returns

PIL.Image Resized image.

secml.data.loader.loader_utils.**crop_img**(*img*, *crop*)

Extract a center crop of the input image.

Parameters

img [PIL.Image.Image] Image to be cropped.

crop [tuple] Dimensions of the desired crop (height, width).

Returns

PIL.Image Cropped image.

Notes

The image center will be computed by rounding the coordinates if necessary. Python round default behavior is toward the closest even decimal.

secml.data.selection

CPrototypesSelector

class secml.data.selection.c_prototypes_selector.**CPrototypesSelector**

Bases: *secml.core.c_creator.CCreator*

Selection of Prototypes.

Prototype selection methods help reducing the number of samples in a dataset by carefully selecting a subset of prototypes.

[1] A good selection strategy should satisfy the following three conditions. First, if some prototypes are similar-that is, if they are close in the space of strings-their distances to a sample string should vary only little. Hence, in this case, some of the respective vector components are redundant. Consequently, a selection algorithm should avoid redundancies. Secondly, to include as much structural information as possible in the prototypes, they should be uniformly distributed over the whole set of patterns. Thirdly, since outliers are likely to introduce noise and distortions, a selection algorithm should disregard outliers.

References

[1]

Attributes

class_type Defines class type.

logger Logger for current object.

sel_idx Returns an array with the indices of the selected prototypes.

verbose Verbosity level of logger output.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
select(self, dataset, n_prototypes)	Selects the prototypes from input dataset.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property sel_idx
Returns an array with the indices of the selected prototypes.

abstract select(*self*, *dataset*, *n_prototypes*)
Selects the prototypes from input dataset.

Parameters

dataset [CDataset] Dataset from which prototypes should be selected

n_prototypes [int] Number of prototypes to be selected.

Returns

reduced_ds [CDataset] Dataset with selected prototypes.

CPSBorder

class secml.data.selection.c_ps_border.**CPSBorder**
Bases: *secml.data.selection.c_prototypes_selector.CPrototypesSelector*

Selection of Prototypes using border strategy.

Selects the prototypes from the borders of the dataset.

References

Spillmann, Barbara, et al. "Transforming strings to vector spaces using prototype selection." Structural, Syntactic, and Statistical Pattern Recognition. Springer Berlin Heidelberg, 2006. 287-296.

Attributes

class_type ['border'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
select(self, dataset, n_prototypes)	Selects the prototypes from input dataset.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

select (*self*, *dataset*, *n_prototypes*)

Selects the prototypes from input dataset.

Parameters

dataset [CDataset] Dataset from which prototypes should be selected

n_prototypes [int] Number of prototypes to be selected.

Returns

reduced_ds [CDataset] Dataset with selected prototypes.

CPSCenter

class secml.data.selection.c_ps_center.**CPSCenter**

Bases: *secml.data.selection.c_prototypes_selector.CPrototypesSelector*

Selection of Prototypes using center strategy.

Selects the prototypes from the center of the dataset.

References

Spillmann, Barbara, et al. "Transforming strings to vector spaces using prototype selection." Structural, Syntactic, and Statistical Pattern Recognition. Springer Berlin Heidelberg, 2006. 287-296.

Attributes

class_type ['center'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
select(self, dataset, n_prototypes)	Selects the prototypes from input dataset.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

select (*self*, *dataset*, *n_prototypes*)

Selects the prototypes from input dataset.

Parameters

dataset [CDataset] Dataset from which prototypes should be selected

n_prototypes [int] Number of prototypes to be selected.

Returns

reduced_ds [CDataset] Dataset with selected prototypes.

CPSKMedians

class secml.data.selection.c_ps_kmedians.**CPSKMedians**

Bases: *secml.data.selection.c_prototypes_selector.CPrototypesSelector*

Selection of Prototypes using K-Medians strategy.

Runs a k-means clustering to obtain a set of clusters from the dataset. Then selects the prototypes as their set medians.

References

Spillmann, Barbara, et al. "Transforming strings to vector spaces using prototype selection." Structural, Syntactic, and Statistical Pattern Recognition. Springer Berlin Heidelberg, 2006. 287-296.

Attributes

class_type ['k-medians'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
select(self, dataset, n_prototypes[, ...])	Selects the prototypes from input dataset.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

select(*self*, *dataset*, *n_prototypes*, *random_state=None*)

Selects the prototypes from input dataset.

Parameters

dataset [CDataset] Dataset from which prototypes should be selected

n_prototypes [int] Number of prototypes to be selected.

random_state [int, RandomState or None, optional] Determines random number generation for centroid initialization. Default None.

Returns

reduced_ds [CDataset] Dataset with selected prototypes.

CPSRandom

class secml.data.selection.c_ps_random.**CPSRandom**

Bases: *secml.data.selection.c_prototypes_selector.CPrototypesSelector*

Selection of Prototypes using random strategy.

Attributes

class_type ['random'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
select(self, dataset, n_prototypes[, ...])	Selects the prototypes from input dataset.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

select(*self*, *dataset*, *n_prototypes*, *random_state=None*)

Selects the prototypes from input dataset.

Parameters

dataset [CDataset] Dataset from which prototypes should be selected

n_prototypes [int] Number of prototypes to be selected.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, is the RandomState instance used by np.random.

Returns

reduced_ds [CDataset] Dataset with selected prototypes.

CPSSpanning

class secml.data.selection.c_ps_spanning.**CPSSpanning**

Bases: *secml.data.selection.c_prototypes_selector.CPrototypesSelector*

Selection of Prototypes using spanning strategy.

Selects the first prototype as the dataset median, and the remaining ones iteratively, by maximizing the distance to the set of previously-selected prototypes.

References

Spillmann, Barbara, et al. "Transforming strings to vector spaces using prototype selection." Structural, Syntactic, and Statistical Pattern Recognition. Springer Berlin Heidelberg, 2006. 287-296.

Attributes

class_type ['spanning'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
select(self, dataset, n_prototypes)	Selects the prototypes from input dataset.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

select(*self*, *dataset*, *n_prototypes*)

Selects the prototypes from input dataset.

Parameters

dataset [CDataset] Dataset from which prototypes should be selected

n_prototypes [int] Number of prototypes to be selected.

Returns

reduced_ds [CDataset] Dataset with selected prototypes.

secml.data.splitter

CDataSplitter

class secml.data.splitter.c_datasplitter.**CDataSplitter**(*num_folds=3*, *random_state=None*)

Bases: *secml.core.c_creator.CCreator*

Abstract class that defines basic methods for dataset splitting.

Parameters

num_folds [int, optional] Number of folds to create. Default 3. This corresponds to the size of tr_idx and ts_idx lists.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, is the RandomState instance used by np.random.

Attributes

class_type Defines class type.

logger Logger for current object.

tr_idx List of training idx obtained with the split of the data.

ts_idx List of test idx obtained with the split of the data.

verbose Verbosity level of logger output.

Methods

compute_indices(self, dataset)	Compute training set and test set indices for each fold.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
split(self, dataset)	Returns a list of split datasets.
timed([msg])	Timer decorator.

abstract compute_indices(*self*, *dataset*)
Compute training set and test set indices for each fold.

Parameters

dataset [CDataset] Dataset to split.

Returns

CDataSplitter Instance of the dataset splitter with tr/ts indices.

split(*self*, *dataset*)
Returns a list of split datasets.

Parameters

dataset [CDataset] Dataset to split.

Returns

split_ds [list of tuple] List of tuples (training set, test set), one for each fold.

property tr_idx
List of training idx obtained with the split of the data.

property ts_idx
List of test idx obtained with the split of the data.

CDataSplitterKFold

class secml.data.splitter.c_datasplitter_kfold.**CDataSplitterKFold**(*num_folds=3*, *random_state=None*)

Bases: *secml.data.splitter.c_datasplitter.CDataSplitter*

K-Folds dataset splitting.

Provides train/test indices to split data in train and test sets. Split dataset into 'num_folds' consecutive folds (with shuffling).

Each fold is then used a validation set once while the k - 1 remaining fold form the training set.

Parameters

num_folds [int, optional] Number of folds to create. Default 3. This correspond to the size of tr_idx and ts_idx lists.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, is the RandomState instance used by np.random.

Examples

```
>>> from secml.data import CDataset
>>> from secml.data.splitter import CDataSplitterKFold
```

```
>>> ds = CDataset([[1,2],[3,4],[5,6]],[1,0,1])
>>> kfold = CDataSplitterKFold(num_folds=3, random_state=0).compute_indices(ds)
>>> print(kfold.num_folds)
3
>>> print(kfold.tr_idx)
[CArray(2,)(dense: [0 1]), CArray(2,)(dense: [0 2]), CArray(2,)(dense: [1 2])]
>>> print(kfold.ts_idx)
[CArray(1,)(dense: [2]), CArray(1,)(dense: [1]), CArray(1,)(dense: [0])]
```

Attributes

class_type ['kfold'] Defines class type.

Methods

compute_indices(self, dataset)	Compute training set and test set indices for each fold.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
split(self, dataset)	Returns a list of split datasets.
timed([msg])	Timer decorator.

compute_indices(*self*, *dataset*)

Compute training set and test set indices for each fold.

Parameters

dataset [CDataset] Dataset to split.

Returns

CDataSplitter Instance of the dataset splitter with tr/ts indices.

CDataSplitterLabelKFold

class secml.data.splitter.c_datasplitter_labelkfold.**CDataSplitterLabelKFold**(*num_folds=3*)

Bases: *secml.data.splitter.c_datasplitter.CDataSplitter*

K-Folds dataset splitting with non-overlapping labels.

The same label will not appear in two different folds (the number of distinct labels has to be at least equal to the number of folds).

The folds are approximately balanced in the sense that the number of distinct labels is approximately the same in each fold.

Parameters

num_folds [int, optional] Number of folds to create. Default 3. This correspond to the size of tr_idx and ts_idx lists.

Examples

```
>>> from secml.data import CDataset
>>> from secml.data import CDataset
>>> from secml.data.splitter import CDataSplitterLabelKFold
>>> ds = CDataset([[1,2],[3,4],[5,6],[7,8]], [1,0,1,2])
>>> kfold = CDataSplitterLabelKFold(num_folds=3).compute_indices(ds)
>>> print(kfold.num_folds)
3
>>> print(kfold.tr_idx)
[CArray(2,)(dense: [1 3]), CArray(3,)(dense: [0 1 2]), CArray(3,)(dense: [0 2 3])]
>>> print(kfold.ts_idx)
[CArray(2,)(dense: [0 2]), CArray(1,)(dense: [3]), CArray(1,)(dense: [1])]
```

Attributes

class_type ['label-kfold'] Defines class type.

Methods

compute_indices(self, dataset)	Compute training set and test set indices for each fold.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.

Continued on next page

Table 36 – continued from previous page

save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
split(self, dataset)	Returns a list of split datasets.
timed([msg])	Timer decorator.

compute_indices(*self*, *dataset*)

Compute training set and test set indices for each fold.

Parameters

dataset [CDataset] Dataset to split.

Returns

CDataSplitter Instance of the dataset splitter with tr/ts indices.

CDataSplitterOpenWorldKFold

class secml.data.splitter.c_datasplitter_openworld.**CDataSplitterOpenWorldKFold**(*num_folds=3*, *n_train_samples=*
n_train_classes=
random_state=None

Bases: *secml.data.splitter.c_datasplitter.CDataSplitter*

Open World K-Folds dataset splitting.

Provides train/test indices to split data in train and test sets.

In an Open World setting, half (or custom number) of the dataset classes are used for training, while all dataset classes are tested.

Split dataset into 'num_folds' consecutive folds (with shuffling).

Each fold is then used a validation set once while the k - 1 remaining fold form the training set.

Parameters

num_folds [int, optional] Number of folds to create. Default 3. This correspond to the size of tr_idx and ts_idx lists.

n_train_samples [int, optional] Number of training samples per client. Default 5.

n_train_classes [int or None] Number of dataset classes to use as training. If not specified half of dataset classes are used (floored).

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, is the RandomState instance used by np.random.

Examples

```
>>> from secml.data import CDataset
>>> from secml.data.splitter import CDataSplitterOpenWorldKFold
```

```
>>> ds = CDataset([[1,2],[3,4],[5,6],[10,20],[30,40],[50,60],
...                [100,200],[300,400]],[1,0,1,2,0,1,0,2])
>>> kfold = CDataSplitterOpenWorldKFold(
...     num_folds=3, n_train_samples=2, random_state=0).compute_indices(ds)
>>> kfold.num_folds
3
>>> print(kfold.tr_idx)
[CArray(2,)(dense: [2 5]), CArray(2,)(dense: [1 4]), CArray(2,)(dense: [0 2])]
>>> print(kfold.ts_idx)
[CArray(6,)(dense: [0 1 3 4 6 7]), CArray(6,)(dense: [0 2 3 5 6 7]), CArray(6,
↪)(dense: [1 3 4 5 6 7])]
>>> print(kfold.tr_classes)  # Class 2 is skipped as there are not enough samples␣
↪(at least 3)
[CArray(1,)(dense: [1]), CArray(1,)(dense: [0]), CArray(1,)(dense: [1])]
```

Attributes

class_type ['open-world-kfold'] Defines class type.

Methods

compute_indices(self, dataset)	Compute training set and test set indices for each fold.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
split(self, dataset)	Returns a list of split datasets.
timed([msg])	Timer decorator.

compute_indices (*self*, *dataset*)

Compute training set and test set indices for each fold.

Parameters

dataset [CDataset] Dataset to split.

Returns

CDataSplitter Instance of the dataset splitter with tr/ts indices.

property tr_classes
List of training classes obtained with the split of the data.

CDataSplitterShuffle

class secml.data.splitter.c_datasplitter_shuffle.**CDataSplitterShuffle**(*num_folds=3*, *train_size=None*, *test_size=None*, *random_state=None*)

Bases: *secml.data.splitter.c_datasplitter.CDataSplitter*

Random permutation dataset splitting.

Yields indices to split data into training and test sets.

Note: contrary to other dataset splitting strategies, random splits do not guarantee that all folds will be different, although this is still very likely for sizeable datasets.

Parameters

num_folds [int, optional] Number of folds to create. Default 3. This correspond to the size of tr_idx and ts_idx lists.

train_size [float, int, or None, optional] If None (default), the value is automatically set to the complement of the test size. If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the train split. If int, represents the absolute number of train samples.

test_size [float, int, or None, optional] If None (default), the value is automatically set to the complement of the train size. If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split. If int, represents the absolute number of test samples.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, is the RandomState instance used by np.random.

Notes

train_size and test_size could not be both None. If one is set to None the other should be a float, representing a percentage, or an integer.

Examples

```
>>> from secml.data import CDataset
>>> from secml.data.splitter import CDataSplitterShuffle
```

```
>>> ds = CDataset([[1,2],[3,4],[5,6]],[1,0,1])
>>> shuffle = CDataSplitterShuffle(num_folds=3, train_size=0.5, random_state=0).
↪compute_indices(ds)
>>> shuffle.num_folds
3
>>> shuffle.tr_idx
[CArray(1,)(dense: [0]), CArray(1,)(dense: [1]), CArray(1,)(dense: [1])]
>>> shuffle.ts_idx
[CArray(2,)(dense: [2 1]), CArray(2,)(dense: [2 0]), CArray(2,)(dense: [0 2])]
```

```
>>> # Setting the train_size or the test_size to an arbitrary percentage
>>> shuffle = CDataSplitterShuffle(num_folds=3, train_size=0.2, random_state=0).
↪compute_indices(ds)
>>> shuffle.num_folds
3
>>> shuffle.tr_idx
[CArray(0,)(dense: []), CArray(0,)(dense: []), CArray(0,)(dense: [])]
>>> shuffle.ts_idx
[CArray(3,)(dense: [2 1 0]), CArray(3,)(dense: [2 0 1]), CArray(3,)(dense: [0 2␣
↪1])]
```

Attributes

class_type ['shuffle'] Defines class type.

Methods

compute_indices(self, dataset)	Compute training set and test set indices for each fold.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.

Continued on next page

Table 38 – continued from previous page

split(self, dataset)	Returns a list of split datasets.
timed([msg])	Timer decorator.

compute_indices (*self*, *dataset*)

Compute training set and test set indices for each fold.

Parameters

dataset [CDataset] Dataset to split.

Returns

CDataSplitter Instance of the dataset splitter with tr/ts indices.

CDataSplitterStratifiedKFold

class secml.data.splitter.c_datasplitter_stratkfold.**CDataSplitterStratifiedKFold**(*num_folds=3*, *random_state=None*)

Bases: *secml.data.splitter.c_datasplitter.CDataSplitter*

Stratified K-Folds dataset splitting.

Provides train/test indices to split data in train test sets.

This dataset splitting object is a variation of KFold, which returns stratified folds. The folds are made by preserving the percentage of samples for each class.

Parameters

num_folds [int, optional] Number of folds to create. Default 3. This correspond to the size of tr_idx and ts_idx lists. For stratified K-Fold, this cannot be higher than the minimum number of samples per class in the dataset.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, is the RandomState instance used by np.random.

Examples

```
>>> from secml.data import CDataset
>>> from secml.data.splitter import CDataSplitterStratifiedKFold
```

```
>>> ds = CDataset([[1,2],[3,4],[5,6],[7,8]],[1,0,0,1])
>>> stratkfold = CDataSplitterStratifiedKFold(num_folds=2, random_state=0).
↪compute_indices(ds)
>>> stratkfold.num_folds  # Cannot be higher than the number of samples per class
2
>>> stratkfold.tr_idx
[CArray(2,)(dense: [1 3]), CArray(2,)(dense: [0 2])]
>>> stratkfold.ts_idx
[CArray(2,)(dense: [0 2]), CArray(2,)(dense: [1 3])]
```

Attributes

class_type ['strat-kfold'] Defines class type.

Methods

compute_indices(self, dataset)	Compute training set and test set indices for each fold.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
split(self, dataset)	Returns a list of split datasets.
timed([msg])	Timer decorator.

compute_indices(*self*, *dataset*)

Compute training set and test set indices for each fold.

Parameters

dataset [CDataset] Dataset to split.

Returns

CDataSplitter Instance of the dataset splitter with tr/ts indices.

CTrainTestSplit

class secml.data.splitter.c_train_test_split.**CTrainTestSplit**(*train_size=None*, *test_size=None*, *random_state=None*, *shuffle=True*)

Bases: *secml.core.c_creator.CCreator*

Train and Test Sets splitter.

Split dataset into random train and test subsets.

Quick utility that wraps CDataSplitterShuffle().compute_indices(ds)) for splitting (and optionally subsampling) data in a oneliner.

Parameters

train_size [float, int, or None, optional] If None (default), the value is automatically set to the complement of the test size. If float, should be between 0.0 and 1.0 and represent the

proportion of the dataset to include in the train split. If int, represents the absolute number of train samples.

test_size [float, int, or None, optional] If None (default), the value is automatically set to the complement of the train size. If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split. If int, represents the absolute number of test samples.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, is the RandomState instance used by np.random.

shuffle [bool, optional] Whether or not to shuffle the data before splitting. If shuffle=False then stratify must be None. Default True.

Notes

train_size and test_size could not be both None. If one is set to None the other should be a float, representing a percentage, or an integer.

Examples

```
>>> from secml.data import CDataset
>>> from secml.data.splitter import CTrainTestSplit
```

```
>>> ds = CDataset([[1,2],[3,4],[5,6],[7,8]],[1,0,1,1])
>>> tr, ts = CTrainTestSplit(train_size=0.5, random_state=0).split(ds)
>>> tr.num_samples
2
>>> ts.num_samples
2
```

```
>>> # Get splitting indices without shuffle
>>> tr_idx, ts_idx = CTrainTestSplit(train_size=0.25,
...     random_state=0, shuffle=False).compute_indices(ds)
>>> tr_idx
CArray(1,)(dense: [0])
>>> ts_idx
CArray(3,)(dense: [1 2 3])
```

```
>>> # At least one sample is needed for each set
>>> tr, ts = CTrainTestSplit(train_size=0.2, random_state=0).split(ds)
Traceback (most recent call last):
    ...
ValueError: train_size should be at least 1 or 0.25
```

Attributes

class_type Defines class type.

logger Logger for current object.

tr_idx Training set indices obtained with the split of the data.

ts_idx Test set indices obtained with the split of the data.

verbose Verbosity level of logger output.

Methods

compute_indices(self, dataset)	Compute training set and test set indices for each fold.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
split(self, dataset)	Split dataset into training set and test set.
timed([msg])	Timer decorator.

compute_indices (*self*, *dataset*)

Compute training set and test set indices for each fold.

Parameters

dataset [CDataset] Dataset to split.

Returns

tr_idx, ts_idx [CArray] Flat arrays with the tr/ts indices.

split (*self*, *dataset*)

Split dataset into training set and test set.

Parameters

dataset [CDataset] Dataset to split.

Returns

ds_train, ds_test [CDataset] Train and Test datasets.

property tr_idx

Training set indices obtained with the split of the data.

property ts_idx

Test set indices obtained with the split of the data.

CChronologicalSplitter

class secml.data.splitter.c_chronological_splitter.**CChronologicalSplitter**(*th_timestamp*, *train_size=1.0*, *test_size=1.0*, *random_state=None*, *shuffle=True*)

Bases: *secml.core.c_creator.CCreator*

Dataset splitter based on timestamps.

Split dataset into train and test subsets, using a timestamp as split point.

A dataset containing *timestamp* and *timestamp_fmt* header attributes is required.

Parameters

th_timestamp [str] The split point in time between training and test set. Samples having *timestamp* <= *th_timestamp* will be put in the training set, while samples with *timestamp* > *th_timestamp* will be used for the test set. The timestamp must follow the ISO 8601 format. Any incomplete timestamp will be parsed too.

train_size [float or int, optional] If float, should be between 0.0 and 1.0 and represent the proportion of the samples having *timestamp* <= *th_timestamp* to include in the train split. Default 1.0. If int, represents the absolute number of train samples.

test_size [float or int, optional] If float, should be between 0.0 and 1.0 and represent the proportion of the samples having *timestamp* > *th_timestamp* to include in the test split. Default 1.0. If int, represents the absolute number of test samples.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, is the RandomState instance used by np.random.

shuffle [bool, optional] Whether or not to shuffle the data before splitting. If shuffle=False then stratify must be None. Default True.

Attributes

class_type Defines class type.

logger Logger for current object.

tr_idx Training set indices obtained with the split of the data.

ts_idx Test set indices obtained with the split of the data.

verbose Verbosity level of logger output.

Methods

compute_indices(self, dataset)	Compute training set and test set indices.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
split(self, dataset)	Split dataset into training set and test set.
timed([msg])	Timer decorator.

compute_indices (*self*, *dataset*)
Compute training set and test set indices.

Parameters

dataset [CDataset] Dataset to split.

Returns

tr_idx, ts_idx [CArray] Flat arrays with the tr/ts indices.

split (*self*, *dataset*)
Split dataset into training set and test set.

Parameters

dataset [CDataset] Dataset to split.

Returns

ds_train, ds_test [CDataset] Train and Test datasets.

property tr_idx
Training set indices obtained with the split of the data.

property ts_idx
Test set indices obtained with the split of the data.

CDataset

class secml.data.c_dataset.**CDataset**(*x*, *y*, *header=None*)

Bases: *secml.core.c_creator.CCreator*

Creates a new dataset.

A dataset consists in a 2-Dimensional patterns array, dense or sparse format, with one pattern for each row and a flat dense array with each pattern's label.

Parameters

x [*array_like* or CArray] Dataset patterns, one for each row. Array is converted to 2-Dimensions before storing.

y [*array_like* or CArray] Dataset labels. Array is converted to dense format and flattened before storing.

header [CDatasetHeader or None, optional] The header for the dataset. Will define any extra parameter. See *CDatasetHeader* docs for more information.

Examples

```
>>> from secml.data import CDataset
```

```
>>> ds = CDataset([[1,2],[3,4],[5,6]],[1,0,1])
>>> print(ds.X)
CArray([[1 2]
 [3 4]
 [5 6]])
>>> print(ds.Y)
CArray([1 0 1])
```

```
>>> ds = CDataset([1,2,3],1)  # Patterns will be converted to 2-Dims
>>> print(ds.X)
CArray([[1 2 3]])
>>> print(ds.Y)
CArray([1])
```

```
>>> from secml.array import CArray
>>> ds = CDataset(CArray([[1,0],[0,4],[1,0]],tosparse=True), CArray([1,0,1],
↪tosparse=True))
>>> print(ds.X)
CArray(  (0, 0)     1
  (1, 1)     4
  (2, 0)     1)
>>> print(ds.Y)
CArray([1 0 1])
```

The number of labels must be equal to the number of samples

```
>>> ds = CDataset([[1,2],[3,4]],1)
Traceback (most recent call last):
 ...
ValueError: number of labels (1) must be equal to the number of samples (2).
```

```
>>> from secml.data import CDatasetHeader
>>> ds = CDataset([1,2,3], 1, CDatasetHeader(id='mydataset', age=34))  # 2 extra␣
↪attributes
>>> print(ds.header.id)
mydataset
>>> print(ds.header.age)
34
```

Attributes

X Dataset Patterns.

Y Dataset Labels.

class_type Defines class type.

classes Classes (unique).

header Dataset header.

isdense True if patterns are stored in dense format, else False.

issparse True if patterns are stored in sparse format, else False.

logger Logger for current object.

num_classes Number of classes.

num_features Number of features.

num_labels Returns dataset's number of labels.

num_samples Number of patterns.

verbose Verbosity level of logger output.

Methods

append(self, dataset)	Append input dataset to current dataset.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_bounds(self[, offset])	Return dataset boundaries plus an offset.
get_class_from_type(class_type)	Return the class associated with input type.
get_labels_onehot(self)	Return dataset labels in one-hot encoding.
get_labels_ovr(self, pos_label)	Return dataset labels in one-vs-rest encoding.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.

Continued on next page

Table 42 – continued from previous page

set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.
todense(self)	Convert dataset's patterns to dense format.
tosparse(self)	Convert dataset's patterns to sparse format.

property X
Dataset Patterns.

property Y
Dataset Labels.

append (*self*, *dataset*)
Append input dataset to current dataset.

Parameters

dataset [CDataset] Dataset to append. Patterns are appended on first axis (axis=0) so the number of features must be equal to dataset.num_features. If current dataset format is sparse, dense dataset to append will be converted to sparse and vice-versa.

Returns

CDataset A new Dataset resulting of appending new data to existing data. Format of resulting dataset is equal to current dataset format.

See also:

CArray.append More information about arrays append.

Notes

Append does not occur in-place: a new dataset is allocated and filled.

Examples

```
>>> from secml.data import CDataset
```

```
>>> ds = CDataset([[1,2],[3,4],[5,6]],[1,0,1])
>>> ds_new = ds.append(CDataset([[10,20],[30,40],[50,60]],[1,0,1]))
>>> print(ds_new.X)
CArray([[ 1  2]
 [ 3  4]
 [ 5  6]
 [10 20]
 [30 40]
 [50 60]])
>>> print(ds_new.Y)
CArray([1 0 1 1 0 1])
```

```
>>> ds_new = ds.append(CDataset([[10,20],[30,40],[50,60]],[1,0,1]).tosparse())
>>> print(ds_new.X)
CArray([[ 1  2]
```

(continues on next page)

(continued from previous page)

```
 [ 3  4]
 [ 5  6]
 [10 20]
 [30 40]
 [50 60]])
>>> print(ds_new.Y)
CArray([1 0 1 1 0 1])
```

property classes
Classes (unique).

get_bounds (*self*, *offset=0.0*)
Return dataset boundaries plus an offset.

Parameters

offset [float] Quantity to be added as an offset. Default 0.

Returns

boundary [list of tuple] Every tuple contain min and max feature value plus an offset for corresponding coordinate.

Examples

```
>>> from secml.array import CArray
>>> from secml.data import  CDataset
```

```
>>> ds = CDataset([[1,2,3],[4,5,6]], [1,2])
>>> ds.get_bounds()
[(1.0, 4.0), (2.0, 5.0), (3.0, 6.0)]
```

get_labels_onehot (*self*)
Return dataset labels in one-hot encoding.

Returns

binary_labels [CArray] A (num_samples, num_classes) array with the dataset labels one-hot encoded.

Examples

```
>>> ds = CDataset([[11,22],[33,44],[55,66],[77,88]], [1,0,2,1])
>>> print(ds.get_labels_onehot())
CArray([[0 1 0]
 [1 0 0]
 [0 0 1]
 [0 1 0]])
```

get_labels_ovr (*self*, *pos_label*)
Return dataset labels in one-vs-rest encoding.

Parameters

pos_label [scalar or str] Label of the class to consider as positive.

Returns

CArray Flat array with 1 when the class label is equal to input positive class's label, else 0.

Examples

```
>>> ds = CDataset([[11,22],[33,44],[55,66],[77,88]], [1,0,2,1])
>>> print(ds.get_labels_ovr(2))
CArray([0 0 1 0])
>>> print(ds.get_labels_ovr(1))
CArray([1 0 0 1])
```

property header
Dataset header.

property isdense
True if patterns are stored in dense format, else False.

property issparse
True if patterns are stored in sparse format, else False.

property num_classes
Number of classes.

property num_features
Number of features.

property num_labels
Returns dataset's number of labels.

property num_samples
Number of patterns.

todense(*self*)
Convert dataset's patterns to dense format.

Returns

CDataset A new CDataset with same patterns converted to dense format. Copy is avoided if possible.

Examples

```
>>> from secml.data import CDataset
```

```
>>> ds = CDataset(CArray([[1,2],[3,4],[5,6]], tosparse=True),[1,0,1]).
↪todense()
>>> print(ds.X)
CArray([[1 2]
 [3 4]
 [5 6]])
```

tosparse(*self*)
Convert dataset's patterns to sparse format.

Returns

CDataset A new CDataset with same patterns converted to sparse format. Copy is avoided if possible.

Examples

```
>>> from secml.data import CDataset
```

```
>>> ds = CDataset([[1,2],[3,4],[5,6]],[1,0,1]).tosparse()
>>> print(ds.X)
CArray(  (0, 0) 1
  (0, 1)        2
  (1, 0)        3
  (1, 1)        4
  (2, 0)        5
  (2, 1)        6)
>>> print(ds.Y)
CArray([1 0 1])
```

CDatasetHeader

class secml.data.c_dataset_header.**CDatasetHeader**(***kwargs*)

Bases: *secml.core.c_creator.CCreator*

Creates a new dataset header.

Parameters to be included into the header could be defined as keyword init arguments or by setting them as new public header attributes.

Immutable objects (scalar, string, tuple, dictionary) will be passed as they are while indexing the header. Arrays will be indexed and the result of indexing will be returned.

To extract a dictionary with the entire set of attributes, use *.get_params()*.

Parameters

kwargs [any, optional] Any extra attribute of the dataset. Could be an immutable object (scalar, tuple, dict, str), or a vector-like CArray. Lists are automatically converted to vector-like CArrays.

Examples

```
>>> from secml.data import CDatasetHeader
>>> from secml.array import CArray
```

```
>>> ds_header = CDatasetHeader(id='mydataset', colors=CArray([1,2,3]))
```

```
>>> print(ds_header.id)
mydataset
>>> print(ds_header.colors)
CArray([1 2 3])
```

```
>>> ds_header.age = 32
>>> print(ds_header.age)
32
```

Attributes

class_type Defines class type.

logger Logger for current object.

num_samples The number of samples for which the header defines extra params.

verbose Verbosity level of logger output.

Methods

append(self, header)	Append input header to current header.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

append(*self*, *header*)

Append input header to current header.

Parameters

header [CDatasetHeader] Header to append. Only attributes which are arrays are merged. Other attributes are set if not already defined in the current header. Otherwise, the value of the attributes in the input header should be equal to the value of the same attribute in the current header.

Returns

CDatasetHeader

See also:

CArray.append More information about arrays append.

Notes

Append does not occur in-place: a new header is allocated and filled.

Examples

```
>>> from secml.data import CDatasetHeader
>>> from secml.array import CArray
```

```
>>> ds_header1 = CDatasetHeader(id={'a': 0, 'b': 2}, a=2, age=CArray([1,2,3]))
>>> ds_header2 = CDatasetHeader(id={'a': 0, 'b': 2}, b=4, age=CArray([1,2,3]))
```

```
>>> ds_merged = ds_header1.append(ds_header2)
>>> ds_merged.age
CArray(6,)(dense: [1 2 3 1 2 3])
>>> ds_merged.id
{'a': 0, 'b': 2}
>>> ds_merged.a
2
>>> ds_merged.b
4
```

property num_samples
The number of samples for which the header defines extra params.

CDatasetPyTorch

class secml.data.c_dataset_pytorch.**CDatasetPyTorch**(*data*, *labels=None*, *transform=None*)

Bases: torch.utils.data.Dataset

CDataset to PyTorch Dataset wrapper.

Parameters

data [CDataset or CArray]

Dataset to be wrapped. Can also be a CArray with the samples and in this case the labels can be passed using the *labels* parameter.

labels [None or CArray] Labels of the dataset. Can be defined if the samples have been passed to the *data* parameter. Input must be a flat array of shape (num_samples,) or a 2-D array with shape (num_samples, num_classes).

transform [torchvision.transforms or None, optional] Transformation(s) to be applied to each ds sample.

Attributes

X

Y

Methods

`__call__`(self, *args, **kwargs)	Call self as a function.

property X

property Y

data_utils

`secml.data.data_utils.`**`label_binarize_onehot`**(*y*)

Return dataset labels in one-hot encoding.

Parameters

y [CArray] Array with the labels to encode. Only integer labels are supported.

Returns

binary_labels [CArray] A (num_samples, num_classes) array with the labels one-hot encoded.

Examples

```
>>> a = CArray([1,0,2,1])
>>> print(label_binarize_onehot(a))
CArray([[0 1 0]
 [1 0 0]
 [0 0 1]
 [0 1 0]])
```

4.7.7 secml.ml

Machine Learning

secml.ml.classifiers

secml.ml.classifiers.multiclass

CClassifierMulticlass

class `secml.ml.classifiers.multiclass.c_classifier_multi.`**`CClassifierMulticlass`**(*classifier*, *preprocess=None*, *n_jobs=1*, ***clf_params*)

Bases: *secml.ml.classifiers.c_classifier.CClassifier*

Generic interface for Multiclass Classifiers.

Parameters

classifier [CClassifier.__class__] Unbound (not initialized) CClassifier subclass.

preprocess [CPreProcess or str or None, optional] Features preprocess to be applied to input data. Can be a CPreProcess subclass or a string with the type of the desired preprocessor. If None, input data is used as is.

n_jobs [int] Number of parallel workers to use for training the classifier. Default 1. Cannot be higher than processor's number of cores.

clf_params [kwargs] Any other construction parameter for the binary classifiers.

Attributes

class_type Defines class type.

classes Return the list of classes on which training has been performed.

classifier Returns the class of the binary classifier used.

logger Logger for current object.

n_classes Number of classes of training dataset.

n_features Number of features (before preprocessing).

n_jobs

num_classifiers Returns the number of instanced binary classifiers.

preprocess Inner preprocessor (if any).

verbose Verbosity level of logger output.

Methods

apply_method(self, method, *args, **kwargs)	Apply input method to all trained classifiers.
backward(self[, w])	Returns the preprocessor gradient wrt data.
binarize_dataset(class_idx, dataset)	Returns the dataset needed by the class_idx binary classifier.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
decision_function(self, x[, y])	Computes the decision function for each pattern in x.
deepcopy(self)	Returns a deep copy of current class.
estimate_parameters(self, dataset, ..., [, ...])	Estimate parameter that give better result respect a chose metric.
fit(self, x, y)	Trains the classifier.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
grad_f_x(self, x, y)	Computes the gradient of the classifier's decision function wrt x.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
is_fitted(self)	Return True if the classifier is trained (fitted).

Continued on next page

Table 45 – continued from previous page

list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
predict(self, x[, return_decision_function])	Perform classification of each pattern in x.
prepare(self, num_classes)	Creates num_classes copies of the binary classifier.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter that has a specific name to a specific value.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

apply_method(*self*, *method*, **args*, ***kwargs*)
Apply input method to all trained classifiers.

Useful to perform a routine after training (e.g. reduction, optim)

method is an unbound method to apply, e.g. CClassiferSVM.set Any other argument for *method* can be passed in.

abstract static binarize_dataset(*class_idx*, *dataset*)
Returns the dataset needed by the class_idx binary classifier.

Parameters

class_idx [int] Index of the target class.

dataset [CDataset] Dataset to binarize.

Returns

bin_dataset [CDataset] Binarized dataset.

property classifier
Returns the class of the binary classifier used.

estimate_parameters(*self*, *dataset*, *parameters*, *splitter*, *metric*, *pick='first'*, *perf_evaluator='xval'*)
Estimate parameter that give better result respect a chose metric.

Parameters

dataset [CDataset] Dataset to be used for evaluating parameters.

parameters [dict] Dictionary with each entry as {parameter: list of values to test}. Example: *{'C': [1, 10, 100], 'gamma': list(10.0 ** CArray.arange(-4, 4))}*

splitter [CDataSplitter or str] Object to use for splitting the dataset into train and validation. A splitter type can be passed as string, in this case all default parameters will be used. For data splitters, num_folds is set to 3 by default. See CDataSplitter docs for more information.

metric [CMetric or str] Object with the metric to use while evaluating the performance. A metric type can be passed as string, in this case all default parameters will be used. See CMetric docs for more information.

pick [{'first', 'last', 'random'}, optional] Defines which of the best parameters set pick. Usually, 'first' correspond to the smallest parameters while 'last' correspond to the biggest. The order is consistent to the parameters dict passed as input.

perf_evaluator [CPerfEvaluator or str, optional] Performance Evaluator to use. Default 'xval'.

Returns

best_parameters [dict] Dictionary of best parameters found through performance evaluation.

get_state(*self*)
Returns the object state dictionary.

Returns

dict Dictionary containing the state of the object. The state of the attributes of binary classifiers is a tuple, with one value for each binary classifier.

property num_classifiers
Returns the number of instanced binary classifiers.

Returns 1 until .fit(dataset) or .prepare(num_classes) is called.

prepare(*self*, *num_classes*)
Creates num_classes copies of the binary classifier.

Creates enough deepcopies of the binary classifier until *num_classes* binary classifiers are instanced. If *num_classes* < *self.num_classifiers*, classifiers in excess are deleted.

Parameters

num_classes [int] Number of binary classifiers to instance.

set(*self*, *param_name*, *param_value*, *copy=False*)
Set a parameter that has a specific name to a specific value.

Only parameters, i.e. PUBLIC or READ/WRITE attributes, can be set.

If setting is performed before training, the parameter to set must be a known *.classifier* attribute or a known attribute of any parameter already set during or after construction.

If possible, a reference to the parameter to set is assigned. Use *copy=True* to always make a deepcopy before set.

Parameters

param_name [str] Name of the parameter to set.

param_value [any] Value to set for the parameter. Using a tuple, one value for each binary classifier can be specified.

copy [bool] By default (False) a reference to the parameter to assign is set. If True or a reference cannot be extracted, a deepcopy of the parameter is done first.

set_state(*self*, *state_dict*, *copy=False*)
Sets the object state using input dictionary.

Only readable attributes of the class, i.e. PUBLIC or READ/WRITE or READ ONLY, can be set.

If possible, a reference to the attribute to set is assigned. Use *copy=True* to always make a deepcopy before set.

Parameters

state_dict [dict] Dictionary containing the state of the object. The state of the attributes of binary classifiers must be specified as a tuple, with one value for each binary classifier.

copy [bool, optional] By default (False) a reference to the attribute to assign is set. If True or a reference cannot be extracted, a deepcopy of the attribute is done first.

property verbose

Verbosity level of logger output.

Available levels are: 0 = no verbose output 1 = info-level logging 2 = debug-level logging

CClassifierMulticlassOVA

class secml.ml.classifiers.multiclass.c_classifier_multi_ova.**CClassifierMulticlassOVA**(*classif*
pre-
pro-
cess=1
n_jobs
***clf_p*

Bases: *secml.ml.classifiers.multiclass.c_classifier_multi.CClassifierMulticlass*, secml.ml.classifiers.gradients.mixin_classifier_gradient.CClassifierGradientMixin

OVA (One-Vs-All) Multiclass Classifier.

Parameters

classifier [unbound class] Unbound (not initialized) CClassifier subclass.

kwargs [any] Any other construction parameter for each OVA classifier.

Attributes

class_type ['ova'] Defines class type.

Methods

apply_method(self, method, *args, **kwargs)	Apply input method to all trained classifiers.
backward(self[, w])	Returns the preprocessor gradient wrt data.
binarize_dataset(class_idx, dataset)	Returns the dataset needed by the class_idx binary classifier.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
decision_function(self, x[, y])	Computes the decision function for each pattern in x.
deepcopy(self)	Returns a deep copy of current class.
estimate_parameters(self, dataset, ... [, ...])	Estimate parameter that give better result respect a chose metric.
fit(self, x, y)	Trains the classifier.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.

Continued on next page

Table 46 – continued from previous page

get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
grad_f_params(self, x, y)	Derivative of the decision function w.r.t.
grad_f_x(self, x, y)	Computes the gradient of the classifier's decision function wrt x.
grad_loss_params(self, x, y[, loss])	Derivative of a given loss w.r.t.
grad_tr_params(self, x, y)	Derivative of the classifier training objective function w.r.t.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
hessian_tr_params(self, x, y)	Hessian of the training objective w.r.t.
is_fitted(self)	Return True if the classifier is trained (fitted).
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
predict(self, x[, return_decision_function])	Perform classification of each pattern in x.
prepare(self, num_classes)	Creates num_classes copies of the binary classifier.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter that has a specific name to a specific value.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

static binarize_dataset (*class_idx*, *dataset*)

Returns the dataset needed by the class_idx binary classifier.

Parameters

class_idx [int] Index of the target class.

dataset [CDataset] Dataset to binarize.

Returns

bin_dataset [CDataset] Binarized dataset.

CClassifierMulticlassOVO

class secml.ml.classifiers.multiclass.c_classifier_multi_ovo.**CClassifierMulticlassOVO**(*classif*
pre-
pro-
cess=
***clf_*

Bases: *secml.ml.classifiers.multiclass.c_classifier_multi.CClassifierMulticlass*, secml.ml.classifiers.gradients.mixin_classifier_gradient.CClassifierGradientMixin

OVO (One-Vs-One) Multiclass Classifier.

Parameters

classifier [unbound class] Unbound (not initialized) CClassifier subclass.

kwargs [any] Any other construction parameter for each OVA classifier.

Attributes

class_type ['ovo'] Defines class type.

Methods

apply_method(self, method, *args, **kwargs)	Apply input method to all trained classifiers.
backward(self[, w])	Returns the preprocessor gradient wrt data.
binarize_dataset(class_idx, dataset)	Returns the dataset needed by the class_idx binary classifier.
binarize_subset(tr_class_idx, vs_class_idx, ...)	Returns the binary dataset tr_class_idx vs vs_class_idx.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
decision_function(self, x[, y])	Computes the decision function for each pattern in x.
deepcopy(self)	Returns a deep copy of current class.
estimate_parameters(self, dataset, ..., [, ...])	Estimate parameter that give better result respect a chose metric.
fit(self, x, y)	Trains the classifier.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
grad_f_params(self, x, y)	Derivative of the decision function w.r.t.
grad_f_x(self, x, y)	Computes the gradient of the classifier's decision function wrt x.
grad_loss_params(self, x, y[, loss])	Derivative of a given loss w.r.t.
grad_tr_params(self, x, y)	Derivative of the classifier training objective function w.r.t.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
hessian_tr_params(self, x, y)	Hessian of the training objective w.r.t.
is_fitted(self)	Return True if the classifier is trained (fitted).
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
predict(self, x[, return_decision_function])	Perform classification of each pattern in x.
prepare(self, num_classes)	Creates num_classes copies of the binary classifier.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter that has a specific name to a specific value.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.

Continued on next page

Table 47 – continued from previous page

set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

static binarize_dataset(*class_idx*, *dataset*)

Returns the dataset needed by the class_idx binary classifier.

Parameters

class_idx [int] Index of the target class.

dataset [CDataset] Dataset to binarize.

Returns

bin_dataset [CDataset] Binarized dataset.

static binarize_subset(*tr_class_idx*, *vs_class_idx*, *dataset*)

Returns the binary dataset tr_class_idx vs vs_class_idx.

Parameters

tr_class_idx [int] Index of the target class.

vs_class_idx: int Index of the opposing class.

dataset [CDataset] Dataset from which the subset should be extracted.

Returns

bin_subset [CDataset] Binarized subset.

property clf_pair_idx

List with the binary classifiers' classes (indices) pairs.

secml.ml.classifiers.reject

CClassifierReject

class secml.ml.classifiers.reject.c_classifier_reject.**CClassifierReject**(*preprocess=None*, *n_jobs=1*)

Bases: *secml.ml.classifiers.c_classifier.CClassifier*

Abstract class that defines basic methods for Classifiers with reject.

A classifier assign a label (class) to new patterns using the information learned from training set.

This interface implements a set of generic methods for training and classification that can be used for every algorithms. However, all of them can be reimplemented if specific routines are needed.

Parameters

preprocess [str or CNormalizer] Features preprocess to applied to input data. Can be a CNormalizer subclass or a string with the desired preprocess type. If None, input data is used as is.

Attributes

class_type Defines class type.

classes Return the list of classes on which training has been performed.

logger Logger for current object.

n_classes Number of classes of training dataset.

n_features Number of features (before preprocessing).

n_jobs

preprocess Inner preprocessor (if any).

verbose Verbosity level of logger output.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
decision_function(self, x[, y])	Computes the decision function for each pattern in x.
deepcopy(self)	Returns a deep copy of current class.
estimate_parameters(self, dataset, ... [, ...])	Estimate parameter that give better result respect a chose metric.
fit(self, x, y)	Trains the classifier.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
grad_f_x(self, x, y)	Computes the gradient of the classifier's decision function wrt x.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
is_fitted(self)	Return True if the classifier is trained (fitted).
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
predict(self, x[, return_decision_function, ...])	Perform classification of each pattern in x.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

abstract predict(*self*, *x*, *return_decision_function=False*, *n_jobs=1*)

Perform classification of each pattern in x.

If a preprocess has been specified, input is normalized before classification.

Parameters

x [CArray] Array with new patterns to classify, 2-Dimensional of shape (n_patterns, n_features).

return_decision_function [bool, optional] Whether to return the decision_function value along with predictions. Default False.

n_jobs [int, optional] Number of parallel workers to use for classification. Default 1. Cannot be higher than processor's number of cores.

Returns

labels [CArray] Flat dense array of shape (n_patterns,) with the label assigned to each test pattern. The classification label is the label of the class associated with the highest score. The rejected samples have label -1.

scores [CArray, optional] Array of shape (n_patterns, n_classes) with classification score of each test pattern with respect to each training class. Will be returned only if *return_decision_function* is True.

CClassifierRejectThreshold

class secml.ml.classifiers.reject.c_classifier_reject_threshold.**CClassifierRejectThreshold**

Bases: *secml.ml.classifiers.reject.c_classifier_reject.CClassifierReject*

Abstract class that defines basic methods for Classifiers with reject based on a certain threshold.

A classifier assign a label (class) to new patterns using the information learned from training set.

The samples for which the higher score is under a certain threshold are rejected by the classifier.

Parameters

clf [CClassifier] Classifier to which we would like to apply a reject threshold. The classifier can also be already fitted.

threshold [float] Rejection threshold.

preprocess [CPreProcess or str or None, optional] Features preprocess to be applied to input data. Can be a CPreProcess subclass or a string with the type of the desired preprocessor. If None, input data is used as is.

Attributes

class_type Defines class type.

classes Return the list of classes on which training has been performed.

clf Returns the inner classifier.

logger Logger for current object.

n_classes Number of classes of training dataset, plus the rejection class.

n_features Number of features (before preprocessing).

n_jobs

preprocess Inner preprocessor (if any).

threshold Returns the rejection threshold.

verbose Verbosity level of logger output.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
decision_function(self, x[, y])	Computes the decision function for each pattern in x.
deepcopy(self)	Returns a deep copy of current class.
estimate_parameters(self, dataset, ... [, ...])	Estimate parameter that give better result respect a chose metric.
fit(self, x, y)	Trains the classifier.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
grad_f_x(self, x, y)	Computes the gradient of the classifier's decision function wrt x.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
is_fitted(self)	Return True if the classifier is trained (fitted).
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
predict(self, x[, return_decision_function, ...])	Perform classification of each pattern in x.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property classes
Return the list of classes on which training has been performed.

property clf
Returns the inner classifier.

property n_classes
Number of classes of training dataset, plus the rejection class.

predict (*self*, *x*, *return_decision_function=False*, *n_jobs=<no value>*)
Perform classification of each pattern in x.

The score matrix of this classifier is equal to the predicted outputs plus a column (corresponding to the reject class) with all its values equal to θ, being θ the reject threshold.

The predicted class is therefore:

$$c = \underset{k}{\operatorname{argmax}} f_k(x)$$

where c correspond to the rejection class (i.e., $c = -1$) only when the maximum taken over the other classes (excluding the reject one) is not greater than the reject threshold θ.

If a preprocess has been specified, input is normalized before classification.

Parameters

x [CArray] Array with new patterns to classify, 2-Dimensional of shape (n_patterns, n_features).

return_decision_function [bool, optional] Whether to return the *decision_function* value along with predictions. Default False.

n_jobs [int, optional] Number of parallel workers to use for classification. Default *_NoValue*. Cannot be higher than processor's number of cores.

Returns

labels [CArray] Flat dense array of shape (n_patterns,) with the label assigned to each test pattern. The classification label is the label of the class associated with the highest score. The samples for which the label is equal -1 are the ones rejected by the classifier

scores [CArray, optional] Array of shape (n_patterns, n_classes) with classification score of each test pattern with respect to each training class. Will be returned only if *return_decision_function* is True.

property threshold

Returns the rejection threshold.

CClassifierDNR

class secml.ml.classifiers.reject.c_classifier_dnr.**CClassifierDNR**(*combiner*, *layer_clf*, *dnn*, *layers*, *threshold*, *n_jobs=1*)

Bases: *secml.ml.classifiers.reject.c_classifier_reject_threshold.CClassifierRejectThreshold*

Deep Neural Rejection (DNR) Classifier. It is composed by a wrapped DNN, one or more layer classifiers trained on inner DNN layer outputs and a combiner trained on layer classifiers scores.

DNR analyzes the representations of input samples at different network layers, and rejects samples which exhibit anomalous behavior with respect to that observed from the training data at such layers.

More details can be found in *tutorials/12-DNR.ipynb* and in:

- https://arxiv.org/pdf/1910.00470.pdf, EURASIP JIS 2020.

Parameters

combiner [CClassifier] The output classifier of DNR. It is trained on layer classifier scores. Its output is thresholded in order to reject samples.

layer_clf [CClassifier or dict] Layer classifier, trained on DNN inner layers outputs. If CClassifier, it is cloned for each selected layer. If dict, it must contain an item for each selected layer as *{'layer_name': CClassifier}*.

dnn [CClassifierDNN] An already trained DNN to be defended.

layers [list of str] Name of one or more DNN layers which outputs will be used to train layer classifiers.

threshold [float] The reject threshold applied to the combiner outputs. If the maximum class score of the combiner is lower than the threshold, the sample is rejected.

n_jobs [int, optional] Number of parallel workers to use for training the classifier. Cannot be higher than processor's number of cores. Default is 1.

Attributes

class_type Defines class type.

classes Return the list of classes on which training has been performed.

clf Returns the inner classifier.

logger Logger for current object.

n_classes Number of classes of training dataset, plus the rejection class.

n_features Number of features (before preprocessing).

n_jobs

preprocess Inner preprocessor (if any).

threshold Returns the rejection threshold.

verbose Verbosity level of logger output.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
compute_threshold(self, rej_percent, ds)	Compute the threshold that must be set in the classifier to have rej_percent rejection rate (accordingly to an estimation on a validation set).
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
decision_function(self, x[, y])	Computes the decision function for each pattern in x.
deepcopy(self)	Returns a deep copy of current class.
estimate_parameters(self, dataset, ..., [, ...])	Estimate parameter that give better result respect a chose metric.
fit(self, x, y)	Trains the classifier.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
grad_f_x(self, x, y)	Computes the gradient of the classifier's decision function wrt x.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
is_fitted(self)	Return True if the classifier is trained (fitted).
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.

Continued on next page

Table 50 – continued from previous page

load_state(self, path)	Sets the object state from file.
predict(self, x[, return_decision_function, ...])	Perform classification of each pattern in x.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

compute_threshold(*self*, *rej_percent*, *ds*)

Compute the threshold that must be set in the classifier to have rej_percent rejection rate (accordingly to an estimation on a validation set).

Parameters

rej_percent [float] Max percentage of rejected samples.

ds [CDataset] Dataset on which the threshold is estimated.

Returns

threshold [float] The estimated reject threshold

secml.ml.classifiers.loss

CLoss

class secml.ml.classifiers.loss.c_loss.**CLoss**

Bases: *secml.core.c_creator.CCreator*

Interface for loss functions.

Attributes

class_type Defines class type.

logger Logger for current object.

suitable_for Defines which problem the loss is suitable for.

verbose Verbosity level of logger output.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
dloss(self, y_true, score)	Computes the derivative of the loss function with respect to *score*.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.

Continued on next page

Table 51 – continued from previous page

get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
loss(self, y_true, score)	Computes the value of the loss function.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

dloss(*self*, *y_true*, *score*)

Computes the derivative of the loss function with respect to *score*.

Parameters

y_true [CArray] Ground truth (correct), targets. Vector-like array.

score [CArray] Outputs (predicted), targets.

Returns

CArray Derivative of the loss function. Vector-like array.

abstract loss(*self*, *y_true*, *score*)

Computes the value of the loss function.

Parameters

y_true [CArray] Ground truth (correct), targets. Vector-like array.

score [CArray] Outputs (predicted), targets.

Returns

CArray Loss function. Vector-like array.

abstract property suitable_for

Defines which problem the loss is suitable for.

Accepted values: - classification - regression

class secml.ml.classifiers.loss.c_loss.**CLossClassification**

Bases: *secml.ml.classifiers.loss.c_loss.CLoss*

Interface for loss functions suitable for classification problems.

Attributes

class_type Defines class type.

logger Logger for current object.

verbose Verbosity level of logger output.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
dloss(self, y_true, score[, pos_label])	Computes the derivative of the loss function with respect to *score*.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
loss(self, y_true, score[, pos_label])	Computes the value of the loss function.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

dloss (*self*, *y_true*, *score*, *pos_label=None*)
Computes the derivative of the loss function with respect to *score*.

Parameters

y_true [CArray] Ground truth (correct), targets. Vector-like array.

score [CArray] Outputs (predicted), targets. 2-D array of shape (n_samples, n_classes) or 1-D flat array of shape (n_samples,).

pos_label [int or None, optional] Default None, meaning that the function derivative is computed for each sample wrt the corresponding true label. Otherwise, this is the class wrt compute the derivative. If *score* is a 1-D flat array, this parameter is ignored.

Returns

CArray Derivative of the loss function. Vector-like array.

abstract loss (*self*, *y_true*, *score*, *pos_label=None*)
Computes the value of the loss function.

Parameters

y_true [CArray] Ground truth (correct), targets. Vector-like array.

score [CArray] Outputs (predicted), targets. 2-D array of shape (n_samples, n_classes) or 1-D flat array of shape (n_samples,).

pos_label [int or None, optional] Default None, meaning that the function is computed for each sample wrt the corresponding true label. Otherwise, this is the class wrt compute the loss function. If *score* is a 1-D flat array, this parameter is ignored.

Returns

CArray Loss function. Vector-like array.

suitable_for = 'classification'

class secml.ml.classifiers.loss.c_loss.**CLossRegression**

Bases: *secml.ml.classifiers.loss.c_loss.CLoss*

Interface for loss functions suitable for regression problems.

Attributes

class_type Defines class type.

logger Logger for current object.

verbose Verbosity level of logger output.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
dloss(self, y_true, score)	Computes the derivative of the loss function with respect to *score*.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
loss(self, y_true, score)	Computes the value of the loss function.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

abstract dloss(*self*, *y_true*, *score*)

Computes the derivative of the loss function with respect to *score*.

Parameters

y_true [CArray] Ground truth (correct), targets. Vector-like array.

score [CArray] Outputs (predicted), targets. Vector-like array of shape (n_samples,).

Returns

CArray Derivative of the loss function. Vector-like array.

abstract loss(*self*, *y_true*, *score*)

Computes the value of the loss function.

Parameters

y_true [CArray] Ground truth (correct), targets. Vector-like array.

score [CArray] Outputs (predicted), targets. Vector-like array of shape (n_samples,).

Returns

CArray Loss function. Vector-like array.

suitable_for = 'regression'

CLossCrossEntropy

class secml.ml.classifiers.loss.c_loss_cross_entropy.**CLossCrossEntropy**

Bases: *secml.ml.classifiers.loss.c_loss.CLossClassification*

Cross Entropy Loss Function (Log Loss).

Cross entropy indicates the distance between what the model believes the output distribution should be, and what the original distribution really is.

The cross entropy loss is defined as (for sample i):

$$L_{\text{cross-entropy}}(y, s) = -\log\left(\frac{e^{s_i}}{\sum_{k=1}^{N} e_k^s}\right)$$

Attributes

class_type ['cross-entropy'] Defines class type.

suitable_for ['classification']

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
dloss(self, y_true, score[, pos_label])	Computes gradient of the Cross Entropy loss w.r.t.the classifier
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
loss(self, y_true, score[, pos_label])	Computes the value of the Cross Entropy loss function.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.

Continued on next page

Table 54 – continued from previous page

`set_state`(self, state_dict[, copy])	Sets the object state using input dictionary.
`timed`([msg])	Timer decorator.

dloss (*self*, *y_true*, *score*, *pos_label=None*)

Computes gradient of the Cross Entropy loss w.r.t.the classifier decision function corresponding to class label pos_label.

Assuming pos_label to be i, the derivative is: p_i - t_i, t_i = 1 if i is equal to y_true_i, 0 otherwise

Then, the elements corresponding to y_true (if pos_label is None) or pos_label will be returned.

Parameters

y_true [CArray] Ground truth (correct), targets. Vector-like array.

score [CArray] Outputs (predicted), targets. 2-D array of shape (n_samples, n_classes).

pos_label [int or None, optional] The class wrt compute the loss function. Default None, meaning that the function is computed for each sample wrt the corresponding true label.

Returns

CArray Loss function. Vector-like array.

loss (*self*, *y_true*, *score*, *pos_label=<no value>*)
Computes the value of the Cross Entropy loss function.

Parameters

y_true [CArray] Ground truth (correct), targets. Vector-like array.

score [CArray] Outputs (predicted), targets. 2-D array of shape (n_samples, n_classes).

Returns

CArray Loss function. Vector-like array.

Notes

Differently from other loss functions, CrossEntropyLoss requires the full array (n_samples, n_classes) of predicted outputs.

CLossEpsilonInsensitive

class secml.ml.classifiers.loss.c_loss_epsilon_insensitive.**CLossEpsilonInsensitive**(*epsilon=0.*
Bases: *secml.ml.classifiers.loss.c_loss.CLossRegression*

Epsilon-Insensitive Loss Function.

Any difference between the current prediction and the ground truth is ignored if is less than the *epsilon* threshold.

Epsilon-Insensitive loss is used by support vector regression.

The Epsilon-Insensitive loss is defined as:

$$L_{\epsilon-\text{ins}}(y, s) = \max\{|y - s| - \epsilon, 0\}$$

Attributes

class_type ['e-insensitive'] Defines class type.

suitable_for ['regression']

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
dloss(self, y_true, score)	Computes the derivative of the epsilon-insensitive loss function
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
loss(self, y_true, score)	Computes the value of the epsilon-insensitive loss function.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

dloss (*self*, *y_true*, *score*)

Computes the derivative of the epsilon-insensitive loss function with respect to *score*.

Parameters

y_true [CArray] Ground truth (correct), targets. Vector-like array.

score [CArray] Outputs (predicted), targets. Vector-like array of shape (n_samples,).

Returns

CArray Derivative of the loss function. Vector-like array.

property epsilon

Threshold parameter epsilon.

loss (*self*, *y_true*, *score*)

Computes the value of the epsilon-insensitive loss function.

Parameters

y_true [CArray] Ground truth (correct), targets. Vector-like array.

score [CArray] Outputs (predicted), targets. Vector-like array of shape (n_samples,).

Returns

CArray Loss function. Vector-like array.

class secml.ml.classifiers.loss.c_loss_epsilon_insensitive.**CLossEpsilonInsensitiveSquared**(

Bases: *secml.ml.classifiers.loss.c_loss_epsilon_insensitive.CLossEpsilonInsensitive*

Squared Epsilon-Insensitive Loss Function.

Any difference between the current prediction and the ground truth is ignored if is less than the *epsilon* threshold.

The Squared Epsilon-Insensitive loss is defined as:

$$L^2_{\epsilon-\text{ins}}(y, s) = \left(\max\left\{|y - s| - \epsilon, 0\right\}\right)^2$$

Attributes

class_type ['e-insensitive-squared'] Defines class type.

suitable_for ['regression']

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
dloss(self, y_true, score)	Computes the derivative of the squared epsilon-insensitive
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
loss(self, y_true, score)	Computes the value of the squared epsilon-insensitive loss function.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

dloss(*self*, *y_true*, *score*)

Computes the derivative of the squared epsilon-insensitive loss function with respect to *score*.

Parameters

y_true [CArray] Ground truth (correct), targets. Vector-like array.

score [CArray] Outputs (predicted), targets. Vector-like array of shape (n_samples,).

Returns

CArray Derivative of the loss function. Vector-like array.

loss (*self*, *y_true*, *score*)
Computes the value of the squared epsilon-insensitive loss function.

Parameters

y_true [CArray] Ground truth (correct), targets. Vector-like array.

score [CArray] Outputs (predicted), targets. Vector-like array of shape (n_samples,).

Returns

CArray Loss function. Vector-like array.

CLossHinge

class secml.ml.classifiers.loss.c_loss_hinge.**CLossHinge**
Bases: *secml.ml.classifiers.loss.c_loss.CLossClassification*

Hinge Loss Function.

The function computes the average distance between the model and the data using hinge loss, a one-sided metric that considers only prediction errors.

Hinge loss is used in maximal margin classifiers such as support vector machines.

After converting the labels to {-1, +1}, then the hinge loss is defined as:

$$L_{\text{Hinge}}(y, s) = \max\{1 - sy, 0\}$$

Attributes

class_type ['hinge'] Defines class type.

suitable_for ['classification']

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
dloss(self, y_true, score[, pos_label])	Computes the derivative of the hinge loss function with respect to *score*.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
loss(self, y_true, score[, pos_label])	Computes the value of the hinge loss function.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.

Continued on next page

Table 57 – continued from previous page

set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

dloss (*self*, *y_true*, *score*, *pos_label=1*)

Computes the derivative of the hinge loss function with respect to *score*.

Parameters

y_true [CArray] Ground truth (correct), targets. Vector-like array.

score [CArray] Outputs (predicted), targets. 2-D array of shape (n_samples, n_classes) or 1-D flat array of shape (n_samples,). If 1-D array, the probabilities provided are assumed to be that of the positive class.

pos_label [{0, 1}, optional] The class wrt compute the loss function derivative. Default 1. If *score* is a 1-D flat array, this parameter is ignored.

Returns

CArray Derivative of the loss function. Vector-like array.

loss (*self*, *y_true*, *score*, *pos_label=1*)

Computes the value of the hinge loss function.

Parameters

y_true [CArray] Ground truth (correct), targets. Vector-like array.

score [CArray] Outputs (predicted), targets. 2-D array of shape (n_samples, n_classes) or 1-D flat array of shape (n_samples,). If 1-D array, the probabilities provided are assumed to be that of the positive class.

pos_label [{0, 1}, optional] The class wrt compute the loss function. Default 1. If *score* is a 1-D flat array, this parameter is ignored.

Returns

CArray Loss function. Vector-like array.

class secml.ml.classifiers.loss.c_loss_hinge.**CLossHingeSquared**

Bases: *secml.ml.classifiers.loss.c_loss.CLossClassification*

Squared Hinge Loss Function.

The function computes the average distance between the model and the data using hinge loss, a one-sided metric that considers only prediction errors.

After converting the labels to {-1, +1}, then the hinge loss is defined as:

$$L^2_{\text{Hinge}}(y, s) = (\max\{1 - sy, 0\})^2$$

Attributes

class_type ['hinge-squared'] Defines class type.

suitable_for ['classification']

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
dloss(self, y_true, score[, pos_label])	Computes the derivative of the squared hinge loss function with respect to *score*.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
loss(self, y_true, score[, pos_label])	Computes the value of the squared hinge loss function.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

dloss(*self*, *y_true*, *score*, *pos_label=1*)

Computes the derivative of the squared hinge loss function with respect to *score*.

Parameters

y_true [CArray] Ground truth (correct), targets. Vector-like array.

score [CArray] Outputs (predicted), targets. 2-D array of shape (n_samples, n_classes) or 1-D flat array of shape (n_samples,). If 1-D array, the probabilities provided are assumed to be that of the positive class.

pos_label [{0, 1}, optional] The class wrt compute the loss function derivative. Default 1. If *score* is a 1-D flat array, this parameter is ignored.

Returns

CArray Derivative of the loss function. Vector-like array.

loss(*self*, *y_true*, *score*, *pos_label=1*)

Computes the value of the squared hinge loss function.

Parameters

y_true [CArray] Ground truth (correct), targets. Vector-like array.

score [CArray] Outputs (predicted), targets. 2-D array of shape (n_samples, n_classes) or 1-D flat array of shape (n_samples,). If 1-D array, the probabilities provided are assumed to be that of the positive class.

pos_label [{0, 1}, optional] The class wrt compute the loss function. Default 1. If *score* is a 1-D flat array, this parameter is ignored.

Returns

CArray Loss function. Vector-like array.

CLossLogistic

class secml.ml.classifiers.loss.c_loss_logistic.**CLossLogistic**

Bases: *secml.ml.classifiers.loss.c_loss.CLossClassification*

Logistic loss function.

Attributes

class_type ['log'] Defines class type.

suitable_for ['classification']

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
dloss(self, y_true, score[, pos_label, bound])	Computes the derivative of the hinge loss function with respect to *score*.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
loss(self, y_true, score[, pos_label, bound])	Computes the value of the logistic loss function.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

dloss(*self*, *y_true*, *score*, *pos_label=1*, *bound=10*)

Computes the derivative of the hinge loss function with respect to *score*.

Parameters

y_true [CArray] Ground truth (correct), targets. Vector-like array.

score [CArray] Outputs (predicted), targets. 2-D array of shape (n_samples, n_classes) or 1-D flat array of shape (n_samples,). If 1-D array, the probabilities provided are assumed to be that of the positive class.

pos_label [{0, 1}, optional] The class wrt compute the loss function derivative. Default 1. If *score* is a 1-D flat array, this parameter is ignored.

bound [scalar or None, optional] Set an upper bound for a linear approximation when -y*s is large to avoid numerical overflows. 10 is a generally acceptable -> log(1+exp(10)) = 10.000045

Returns

CArray Derivative of the loss function. Vector-like array.

loss (*self*, *y_true*, *score*, *pos_label=1*, *bound=10*)
Computes the value of the logistic loss function.

Parameters

y_true [CArray] Ground truth (correct), targets. Vector-like array.

score [CArray] Outputs (predicted), targets. 2-D array of shape (n_samples, n_classes) or 1-D flat array of shape (n_samples,). If 1-D array, the probabilities provided are assumed to be that of the positive class.

pos_label [{0, 1}, optional] The class wrt compute the loss function. Default 1. If *score* is a 1-D flat array, this parameter is ignored.

bound [scalar or None, optional] Set an upper bound for a linear approximation when -y*s is large to avoid numerical overflows. 10 is a generally acceptable -> log(1+exp(10)) = 10.000045

Returns

CArray Loss function. Vector-like array.

CLossSquare

class secml.ml.classifiers.loss.c_loss_squared.**CLossQuadratic**
Bases: *secml.ml.classifiers.loss.c_loss.CLossRegression*

Quadratic Loss Function (Ordinary Least Squares).

The quadratic loss is defined as:

$$L_{\text{Quadratic}}(y, s) = \frac{1}{2}(s - y)^2$$

Attributes

class_type ['quadratic'] Defines class type.

suitable_for ['regression']

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
dloss(self, y_true, score)	Computes the derivative of the quadratic loss function with respect to *score*.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.

Continued on next page

Table 60 – continued from previous page

get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
loss(self, y_true, score)	Computes the value of the quadratic loss function.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

dloss (*self*, *y_true*, *score*)
Computes the derivative of the quadratic loss function with respect to *score*.

Parameters

y_true [CArray] Ground truth (correct), targets. Vector-like array.

score [CArray] Outputs (predicted), targets. Vector-like array of shape (n_samples,).

Returns

CArray Derivative of the loss function. Vector-like array.

loss (*self*, *y_true*, *score*)
Computes the value of the quadratic loss function.

Parameters

y_true [CArray] Ground truth (correct), targets. Vector-like array.

score [CArray] Outputs (predicted), targets. Vector-like array of shape (n_samples,).

Returns

CArray Loss function. Vector-like array.

class secml.ml.classifiers.loss.c_loss_squared.**CLossSquare**
Bases: *secml.ml.classifiers.loss.c_loss.CLossClassification*

Square Loss Function.

The square loss is defined as:

$$L_{\text{Square}}(y, s) = (1 - sy)$$

Attributes

class_type ['square'] Defines class type.

suitable_for ['classification']

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
dloss(self, y_true, score[, pos_label])	Computes the derivative of the square loss function with respect to *score*.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
loss(self, y_true, score[, pos_label])	Computes the value of the squared epsilon-insensitive loss function.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

dloss(*self*, *y_true*, *score*, *pos_label=1*)

Computes the derivative of the square loss function with respect to *score*.

Parameters

y_true [CArray] Ground truth (correct), targets. Vector-like array.

score [CArray] Outputs (predicted), targets. 2-D array of shape (n_samples, n_classes) or 1-D flat array of shape (n_samples,). If 1-D array, the probabilities provided are assumed to be that of the positive class.

pos_label [{0, 1}, optional] The class wrt compute the loss function derivative. Default 1. If *score* is a 1-D flat array, this parameter is ignored.

Returns

CArray Derivative of the loss function. Vector-like array.

loss(*self*, *y_true*, *score*, *pos_label=1*)

Computes the value of the squared epsilon-insensitive loss function.

Parameters

y_true [CArray] Ground truth (correct), targets. Vector-like array.

score [CArray] Outputs (predicted), targets. 2-D array of shape (n_samples, n_classes) or 1-D flat array of shape (n_samples,). If 1-D array, the probabilities provided are assumed to be that of the positive class.

pos_label [{0, 1}, optional] The class wrt compute the loss function. Default 1. If *score* is a 1-D flat array, this parameter is ignored.

Returns

CArray Loss function. Vector-like array.

CSoftmax

class secml.ml.classifiers.loss.c_softmax.**CSoftmax**

Bases: *secml.core.c_creator.CCreator*

Softmax function.

Attributes

class_type Defines class type.

logger Logger for current object.

verbose Verbosity level of logger output.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
gradient(self, s, y)	Gradient of the softmax function.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
softmax(self, s)	Apply the softmax function to input.
timed([msg])	Timer decorator.

gradient (*self*, *s*, *y*)

Gradient of the softmax function.

The derivative of the y-th output of the softmax function w.r.t. all the inputs is given by:

$$\left[\frac{s'_y}{a'_1}, \ldots, \frac{s'_y}{a'_n}\right]$$

$$where: \frac{s'_y}{a'_i} = s_y(\delta - s_i)$$

$$with: \delta = 1 if i = j \delta = 0 if i \neq j$$

Parameters

s [CArray] 2-D array of shape (1, n_classes) with input data.

y [int] The class wrt compute the gradient.

Returns

CArray Softmax function gradient. Vector-like array.

softmax (*self*, *s*)

Apply the softmax function to input.

The softmax function is defined for the vector *s* and for the i-th class as:

$$\text{SoftMax}(y, s) = [a_1, \ldots, a_n] -> [s_1, \ldots, s_n]$$
$$where : \mathrm{s}_y = \frac{e^{a_j}}{\sum_{i=1}^{N} e_i^a} \forall 1 = 1..N$$

Parameters

s [CArray] 2-D array of shape (n_samples, n_classes) with input data.

Returns

CArray Softmax function. Same shape of input array.

Examples

```
>>> from secml.array import CArray
>>> from secml.ml.classifiers.loss import CSoftmax
```

```
>>> a = CArray([[1, 2, 3], [2, 4, 5]])
>>> print(CSoftmax().softmax(a))
CArray([[0.090031 0.244728 0.665241]
 [0.035119 0.259496 0.705385]])
```

secml.ml.classifiers.regularizer

CRegularizer

class secml.ml.classifiers.regularizer.c_regularizer.**CRegularizer**

Bases: *secml.core.c_creator.CCreator*

Abstract class that defines basic methods for regularizer functions.

Attributes

class_type Defines class type.

logger Logger for current object.

verbose Verbosity level of logger output.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
dregularizer(self, *args, **kwargs)	Gets the derivative of regularizer.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
regularizer(self, *args, **kwargs)	Gets value of regularizer.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

dregularizer (*self*, **args*, ***kwargs*)
Gets the derivative of regularizer.

abstract regularizer (*self*, **args*, ***kwargs*)
Gets value of regularizer.

CRegularizerElasticNet

class secml.ml.classifiers.regularizer.c_regularizer_elastic_net.**CRegularizerElasticNet**(*l1_*
Bases: *secml.ml.classifiers.regularizer.c_regularizer.CRegularizer*

ElasticNet Regularizer.

A convex combination of L2 and L1, where ρ is given by *1 - l1_ratio*.

ElasticNet Regularizer is given by:

$$R(w) := \frac{\rho}{2} \sum_{i=1}^{n} w_i^2 + (1 - \rho) \sum_{i=1}^{n} |w_i|$$

Attributes

class_type ['elastic-net'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
dregularizer(self, w)	Returns the derivative of the elastic-net regularizer
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
regularizer(self, w)	Returns ElasticNet Regularizer.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

dregularizer (*self*, *w*)
Returns the derivative of the elastic-net regularizer

Parameters

w [CArray] Vector-like array.

property l1_ratio
Get l1-ratio.

regularizer (*self*, *w*)
Returns ElasticNet Regularizer.

Parameters

w [CArray] Vector-like array.

CRegularizerL1

class secml.ml.classifiers.regularizer.c_regularizer_l1.**CRegularizerL1**
Bases: *secml.ml.classifiers.regularizer.c_regularizer.CRegularizer*

Norm-L1 Regularizer.

This function leads to sparse solutions.

L1 Regularizer is given by:

$$R(w) := \sum_{i=1}^{n} |w_i|$$

Attributes

class_type ['l1'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
dregularizer(self, w)	Returns Norm-L1 derivative.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
regularizer(self, w)	Returns Norm-L1.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

dregularizer(*self*, *w*)
Returns Norm-L1 derivative.

Parameters

w [CArray] Vector-like array.

regularizer(*self*, *w*)
Returns Norm-L1.

Parameters

w [CArray] Vector-like array.

CRegularizerL2

class secml.ml.classifiers.regularizer.c_regularizer_l2.**CRegularizerL2**
Bases: *secml.ml.classifiers.regularizer.c_regularizer.CRegularizer*

Norm-L2 Regularizer.

L2 Regularizer is given by:

$$R(w) := \frac{1}{2} \sum_{i=1}^{n} w_i^2$$

Attributes

class_type ['l2'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
dregularizer(self, w)	Return Norm-L2 derivative.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
regularizer(self, w)	Returns Norm-L2.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

dregularizer(*self*, *w*)
Return Norm-L2 derivative.

Parameters

w [CArray] Vector-like array.

regularizer(*self*, *w*)
Returns Norm-L2.

Parameters

w [CArray] Vector-like array.

CClassifier

class secml.ml.classifiers.c_classifier.**CClassifier**(*preprocess=None*, *n_jobs=1*)
Bases: secml.ml.c_module.CModule

Abstract class that defines basic methods for Classifiers.

A classifier assign a label (class) to new patterns using the information learned from training set.

This interface implements a set of generic methods for training and classification that can be used for every algorithms. However, all of them can be reimplemented if specific routines are needed.

Parameters

preprocess [CPreProcess or str or None, optional] Features preprocess to be applied to input data. Can be a CPreProcess subclass or a string with the type of the desired preprocessor. If None, input data is used as is.

n_jobs [int, optional] Number of parallel workers to use for training the classifier. Cannot be higher than processor's number of cores. Default is 1.

Attributes

class_type Defines class type.

classes Return the list of classes on which training has been performed.

logger Logger for current object.

n_classes Number of classes of training dataset.

n_features Number of features (before preprocessing).

n_jobs

preprocess Inner preprocessor (if any).

verbose Verbosity level of logger output.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
decision_function(self, x[, y])	Computes the decision function for each pattern in x.
deepcopy(self)	Returns a deep copy of current class.
estimate_parameters(self, dataset, ... [, ...])	Estimate parameter that give better result respect a chose metric.
fit(self, x, y)	Trains the classifier.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
grad_f_x(self, x, y)	Computes the gradient of the classifier's decision function wrt x.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
is_fitted(self)	Return True if the classifier is trained (fitted).
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
predict(self, x[, return_decision_function])	Perform classification of each pattern in x.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.

Continued on next page

Table 67 – continued from previous page

`timed`([msg])	Timer decorator.

property classes
Return the list of classes on which training has been performed.

decision_function(*self*, *x*, *y=None*)
Computes the decision function for each pattern in x.

If a preprocess has been specified, input is normalized before computing the decision function.

Note: The actual decision function should be implemented inside `_decision_function` method.

Parameters

x [CArray] Array with new patterns to classify, 2-Dimensional of shape (n_patterns, n_features).

y [int or None, optional] The label of the class wrt the function should be calculated. If None, return the output for all classes.

Returns

score [CArray] Value of the decision function for each test pattern. Dense flat array of shape (n_samples,) if y is not None, otherwise a (n_samples, n_classes) array.

estimate_parameters(*self*, *dataset*, *parameters*, *splitter*, *metric*, *pick='first'*, *perf_evaluator='xval'*)
Estimate parameter that give better result respect a chose metric.

Parameters

dataset [CDataset] Dataset to be used for evaluating parameters.

parameters [dict] Dictionary with each item as *{parameter: list of values to test}*. Example: *{'C': [1, 10, 100], 'gamma': list(10.0 ** CArray.arange(-4, 4))}*

splitter [CDataSplitter or str] Object to use for splitting the dataset into train and validation. A splitter type can be passed as string, in this case all default parameters will be used. For data splitters, num_folds is set to 3 by default. See CDataSplitter docs for more information.

metric [CMetric or str] Object with the metric to use while evaluating the performance. A metric type can be passed as string, in this case all default parameters will be used. See CMetric docs for more information.

pick [{'first', 'last', 'random'}, optional] Defines which of the best parameters set pick. Usually, 'first' correspond to the smallest parameters while 'last' correspond to the biggest. The order is consistent to the parameters dict passed as input.

perf_evaluator [CPerfEvaluator or str, optional] Performance Evaluator to use. Default 'xval'.

Returns

best_parameters [dict] Dictionary of best parameters found through performance evaluation.

fit(*self*, *x*, *y*)
Trains the classifier.

If a preprocess has been specified, input is normalized before training.

For multiclass case see *.CClassifierMulticlass*.

Parameters

x [CArray] Array to be used for training with shape (n_samples, n_features).

y [CArray or None, optional] Array of shape (n_samples,) containing the class labels. Can be None if not required by the algorithm.

Returns

CClassifier Trained classifier.

fit_forward(*self*, *x*, *y=None*, *caching=False*)

Fit estimator using data and then execute forward on the data.

To avoid returning over-fitted scores on the training set, this method runs a 5-fold cross validation on training data and returns the validation scores.

Parameters

x [CArray] Array with shape (n_samples, n_features) to be transformed and to be used for training.

y [CArray or None, optional] Array of shape (n_samples,) containing the class labels. Can be None if not required by the algorithm.

caching: bool True if preprocessed x should be cached for backward pass

Returns

CArray Transformed input data.

See also:

fit fit the preprocessor.

forward run forward function on input data.

grad_f_x(*self*, *x*, *y*)

Computes the gradient of the classifier's decision function wrt x.

Parameters

x [CArray or None, optional] The input point. The gradient will be computed at x.

y [int] Binary index of the class wrt the gradient must be computed.

Returns

gradient [CArray] The gradient of the linear classifier's decision function wrt decision function input. Vector-like array.

is_fitted(*self*)

Return True if the classifier is trained (fitted).

Returns

bool True or False depending on the result of the call to *check_is_fitted*.

property n_classes

Number of classes of training dataset.

property n_features

Number of features (before preprocessing).

predict (*self*, *x*, *return_decision_function=False*)
Perform classification of each pattern in x.

If preprocess has been specified, input is normalized before classification.

Parameters

x [CArray] Array with new patterns to classify, 2-Dimensional of shape (n_patterns, n_features).

return_decision_function [bool, optional] Whether to return the *decision_function* value along with predictions. Default False.

Returns

labels [CArray] Flat dense array of shape (n_patterns,) with the label assigned to each test pattern. The classification label is the label of the class associated with the highest score.

scores [CArray, optional] Array of shape (n_patterns, n_classes) with classification score of each test pattern with respect to each training class. Will be returned only if *return_decision_function* is True.

CClassifierLinear

class secml.ml.classifiers.c_classifier_linear.**CClassifierLinearMixin**
Bases: object

Mixin class that defines basic methods for linear classifiers.

A linear classifier assigns a label (class) to new patterns computing the inner product between the patterns and a vector of weights for each training set feature.

This interface defines the weight and bias, and the forward and backward functions for linear classifiers.

Attributes

b Bias term.

w Vector with feature weights (dense or sparse).

abstract property b
Bias term.

abstract property w
Vector with feature weights (dense or sparse).

CClassifierSkLearn

class secml.ml.classifiers.sklearn.c_classifier_sklearn.**CClassifierSkLearn**(*sklearn_model*, *preprocess=None*)
Bases: *secml.ml.classifiers.sklearn.c_classifier_sklearn.CWrapperSkLearnMixin*, *secml.ml.classifiers.c_classifier.CClassifier*

Generic wrapper for SkLearn classifiers.

Parameters

sklearn_model [*sklearn.base.BaseEstimator* object] The scikit-learn model to wrap. Must implement *fit* and either *decision_function* or *predict_proba* methods.

preprocess [CModule or str or None, optional] Features preprocess to be applied to input data. Can be a CModule subclass or a string with the type of the desired preprocessor. If None, input data is used as is.

Attributes

class_type ['sklearn-clf'] Defines class type.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
decision_function(self, x[, y])	Computes the decision function for each pattern in x.
deepcopy(self)	Returns a deep copy of current class.
estimate_parameters(self, dataset, ..., [, ...])	Estimate parameter that give better result respect a chose metric.
fit(self, x, y)	Trains the classifier.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class and SkLearn model parameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
grad_f_x(self, x, y)	Computes the gradient of the classifier's decision function wrt x.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
is_fitted(self)	Return True if the classifier is trained (fitted).
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
predict(self, x[, return_decision_function])	Perform classification of each pattern in x.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

class secml.ml.classifiers.sklearn.c_classifier_sklearn.**CWrapperSkLearnMixin**(*sklearn_model*)

Bases: object

Generic wrapper for SkLearn instances.

Parameters

sklearn_model [*sklearn.base.BaseEstimator* object] The scikit-learn instance to wrap.

Attributes

sklearn_model Wrapped SkLearn classifier.

Methods

get_params(self)	Returns the dictionary of class and SkLearn model parameters.

get_params (*self*)
Returns the dictionary of class and SkLearn model parameters.

A parameter is a PUBLIC or READ/WRITE attribute.

property sklearn_model
Wrapped SkLearn classifier.

CClassifierDecisionTree

class secml.ml.classifiers.sklearn.c_classifier_decision_tree.**CClassifierDecisionTree**(*criteri*
split-
ter='b
max_d
min_s
ran-
dom_s
pre-
pro-
cess=

Bases: *secml.ml.classifiers.sklearn.c_classifier_sklearn.CClassifierSkLearn*

Decision Tree Classifier.

Parameters

criterion [str, optional] The function to measure the quality of a split. Supported criteria are 'gini' (default) for the Gini impurity and 'entropy' for the information gain.

splitter [str, optional] The strategy used to choose the split at each node. Supported strategies are 'best' (default) to choose the best split and 'random' to choose the best random split.

max_depth [int or None, optional] The maximum depth of the tree. If None (default), then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split [int or float, optional] The minimum number of samples required to split an internal node. If int, then consider *min_samples_split* as the minimum number. If float, then *min_samples_split* is a fraction and *ceil(min_samples_split * n_samples)* are the minimum number of samples for each split. Default 2.

random_state [int, RandomState or None, optional] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*. Default None.

preprocess [CPreProcess or str or None, optional] Features preprocess to be applied to input data. Can be a CPreProcess subclass or a string with the type of the desired preprocessor. If None, input data is used as is.

Attributes

class_type ['dec-tree'] Defines class type.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
decision_function(self, x[, y])	Computes the decision function for each pattern in x.
deepcopy(self)	Returns a deep copy of current class.
estimate_parameters(self, dataset, ..., [, ...])	Estimate parameter that give better result respect a chose metric.
fit(self, x, y)	Trains the classifier.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class and SkLearn model parameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
grad_f_x(self, x, y)	Computes the gradient of the classifier's decision function wrt x.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
is_fitted(self)	Return True if the classifier is trained (fitted).
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
predict(self, x[, return_decision_function])	Perform classification of each pattern in x.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

CClassifierKNN

class secml.ml.classifiers.sklearn.c_classifier_knn.**CClassifierKNN**(*n_neighbors=5*, *weights='uniform'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *metric_params=None*, *preprocess=None*)

Bases: *secml.ml.classifiers.sklearn.c_classifier_sklearn.CClassifierSkLearn*

K Neighbors Classifiers.

Parameters

n_neighbors [int, optional] Number of neighbors to use by default for *kneighbors* queries. Default 5.

weights [str or callable, optional] Weight function used in prediction. If 'uniform' (default), all points in each neighborhood are weighted equally; if 'distance' points are weighted by the inverse of their distance. Can also be an user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

algorithm [{'auto', 'ball_tree', 'kd_tree', 'brute'}, optional] Algorithm used to compute the nearest neighbors. If 'auto' (default), the most appropriate algorithm is decided based on the values passed to `fit` method.

leaf_size [int, optional] Leaf size passed to BallTree or KDTree. Default 30.

p [int, optional] Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2 (default). For arbitrary p, minkowski_distance (l_p) is used.

metric [str or callable, optional] The distance metric to use for the tree. If 'minkowski' (default) and p = 2, it is equivalent to the standard Euclidean metric. If metric is 'precomputed', X is assumed to be a distance matrix and must be square during fit.

metric_params [dict, optional] Additional keyword arguments for the metric function.

preprocess [CPreProcess or str or None, optional] Features preprocess to be applied to input data. Can be a CPreProcess subclass or a string with the type of the desired preprocessor. If None, input data is used as is.

Attributes

class_type ['knn'] Defines class type.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
decision_function(self, x[, y])	Computes the decision function for each pattern in x.
deepcopy(self)	Returns a deep copy of current class.
estimate_parameters(self, dataset, ... [, ...])	Estimate parameter that give better result respect a chose metric.
fit(self, x, y)	Trains the classifier.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class and SkLearn model parameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
grad_f_x(self, x, y)	Computes the gradient of the classifier's decision function wrt x.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
is_fitted(self)	Return True if the classifier is trained (fitted).
kneighbors(self, x[, num_samples])	Find the training samples nearest to x
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
predict(self, x[, return_decision_function])	Perform classification of each pattern in x.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

kneighbors(*self*, *x*, *num_samples=None*)

Find the training samples nearest to x

Parameters

x [CArray] The query point or points.

num_samples: int or None Number of neighbors to get. if None, use n_neighbors

Returns

dist [CArray] Array representing the lengths to points

index_point: CArray Indices of the nearest points in the training set

tr_dataset.X: CArray Training samples

property tr
Training set.

CClassifierLogistic

class secml.ml.classifiers.sklearn.c_classifier_logistic.**CClassifierLogistic**(*C=1.0*, *max_iter=100*, *random_state=None*, *preprocess=None*)

Bases: *secml.ml.classifiers.c_classifier_linear.CClassifierLinearMixin*, *secml.ml.classifiers.sklearn.c_classifier_sklearn.CClassifierSkLearn*, secml.ml.classifiers.gradients.mixin_classifier_gradient_logistic.CClassifierGradientLogisticMixin

Logistic Regression (aka logit, MaxEnt) classifier.

Parameters

C [float, optional] Penalty parameter C of the error term. Default 1.0.

max_iter [int, optional] Maximum number of iterations taken for the solvers to converge. Default 100.

random_state [int, RandomState or None, optional] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*. Default None.

preprocess [CPreProcess or str or None, optional] Features preprocess to be applied to input data. Can be a CPreProcess subclass or a string with the type of the desired preprocessor. If None, input data is used as is.

Attributes

class_type ['logistic'] Defines class type.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
decision_function(self, x[, y])	Computes the decision function for each pattern in x.
deepcopy(self)	Returns a deep copy of current class.
estimate_parameters(self, dataset, ..., [, ...])	Estimate parameter that give better result respect a chose metric.
fit(self, x, y)	Trains the classifier.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.

Continued on next page

Table 72 – continued from previous page

forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class and SkLearn model parameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
grad_f_params(self, x[, y])	Derivative of the decision function w.r.t.
grad_f_x(self, x, y)	Computes the gradient of the classifier's decision function wrt x.
grad_loss_params(self, x, y[, loss])	Derivative of the classifier loss w.r.t.
grad_tr_params(self, x, y)	Derivative of the classifier training objective w.r.t. the classifier
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
hessian_tr_params(self, x, y)	Hessian of the training objective w.r.t.
is_fitted(self)	Return True if the classifier is trained (fitted).
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
predict(self, x[, return_decision_function])	Perform classification of each pattern in x.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property b
Bias term.

property w
Vector with feature weights (dense or sparse).

CClassifierNearestCentroid

class secml.ml.classifiers.sklearn.c_classifier_nearest_centroid.**CClassifierNearestCentroid**

Bases: *secml.ml.classifiers.sklearn.c_classifier_sklearn.CClassifierSkLearn*

CClassifierNearestCentroid.

Parameters

metric [str or callable, optional] The metric to use when calculating distance between instances in a feature array. Default 'euclidean'.

shrink_threshold [float, optional] Threshold for shrinking centroids to remove features.

preprocess [CPreProcess or str or None, optional] Features preprocess to be applied to input data. Can be a CPreProcess subclass or a string with the type of the desired preprocessor. If

None, input data is used as is.

Attributes

class_type ['nrst-centroid'] Defines class type.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
decision_function(self, x[, y])	Computes the decision function for each pattern in x.
deepcopy(self)	Returns a deep copy of current class.
estimate_parameters(self, dataset, ... [, ...])	Estimate parameter that give better result respect a chose metric.
fit(self, x, y)	Trains the classifier.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class and SkLearn model parameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
grad_f_x(self, x, y)	Computes the gradient of the classifier's decision function wrt x.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
is_fitted(self)	Return True if the classifier is trained (fitted).
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
predict(self, x[, return_decision_function])	Perform classification of each pattern in x.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property centroids

property metric

CClassifierRandomForest

class secml.ml.classifiers.sklearn.c_classifier_random_forest.**CClassifierRandomForest**(*n_estim*
criterion='
max_d
min_sa
random_s
preprocess=l

Bases: *secml.ml.classifiers.sklearn.c_classifier_sklearn.CClassifierSkLearn*

Random Forest classifier.

Parameters

n_estimators [int, optional] The number of trees in the forest. Default 10.

criterion [str, optional] The function to measure the quality of a split. Supported criteria are 'gini' (default) for the Gini impurity and 'entropy' for the information gain.

max_depth [int or None, optional] The maximum depth of the tree. If None (default), then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.

min_samples_split [int or float, optional] The minimum number of samples required to split an internal node. If int, then consider *min_samples_split* as the minimum number. If float, then *min_samples_split* is a fraction and *ceil(min_samples_split * n_samples)* are the minimum number of samples for each split. Default 2.

random_state [int, RandomState or None, optional] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*. Default None.

preprocess [CPreProcess or str or None, optional] Features preprocess to be applied to input data. Can be a CPreProcess subclass or a string with the type of the desired preprocessor. If None, input data is used as is.

Attributes

class_type ['random-forest'] Defines class type.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
decision_function(self, x[, y])	Computes the decision function for each pattern in x.
deepcopy(self)	Returns a deep copy of current class.

Continued on next page

Table 74 – continued from previous page

estimate_parameters(self, dataset, ... [, ...])	Estimate parameter that give better result respect a chose metric.
fit(self, x, y)	Trains the classifier.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class and SkLearn model parameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
grad_f_x(self, x, y)	Computes the gradient of the classifier's decision function wrt x.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
is_fitted(self)	Return True if the classifier is trained (fitted).
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
predict(self, x[, return_decision_function])	Perform classification of each pattern in x.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

CClassifierRidge

class secml.ml.classifiers.sklearn.c_classifier_ridge.**CClassifierRidge**(*alpha=1.0*, *max_iter=100000.0*, *class_weight=None*, *tol=0.0001*, *fit_intercept=True*, *preprocess=None*)

Bases: *secml.ml.classifiers.c_classifier_linear.CClassifierLinearMixin*, *secml.ml.classifiers.sklearn.c_classifier_sklearn.CClassifierSkLearn*, secml.ml.classifiers.gradients.mixin_classifier_gradient_ridge.CClassifierGradientRidgeMixin

Ridge Classifier.

Parameters

alpha [float, optional] Regularization strength; must be a positive float. Regularization improves the conditioning of the problem and reduces the variance of the estimates. Larger values specify stronger regularization. Default 1.0.

max_iter [int, optional] Maximum number of iterations for conjugate gradient solver. Default 1e5.

class_weight [{dict, 'balanced', None}, optional] Set the parameter C of class i to *class_weight[i] * C*. If not given (default), all classes are supposed to have weight one. The 'balanced' mode uses the values of labels to automatically adjust weights inversely proportional to class frequencies as *n_samples / (n_classes * np.bincount(y))*.

tol [float, optional] Precision of the solution. Default 1e-4.

fit_intercept [bool, optional] If True (default), the intercept is calculated, else no intercept will be used in calculations (e.g. data is expected to be already centered).

preprocess [CPreProcess or str or None, optional] Features preprocess to be applied to input data. Can be a CPreProcess subclass or a string with the type of the desired preprocessor. If None, input data is used as is.

Attributes

class_type ['ridge'] Defines class type.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
decision_function(self, x[, y])	Computes the decision function for each pattern in x.
deepcopy(self)	Returns a deep copy of current class.
estimate_parameters(self, dataset, ... [, ...])	Estimate parameter that give better result respect a chose metric.
fit(self, x, y)	Trains the classifier.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class and SkLearn model parameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
grad_f_params(self, x[, y])	Derivative of the decision function w.r.t.
grad_f_x(self, x, y)	Computes the gradient of the classifier's decision function wrt x.
grad_loss_params(self, x, y[, loss])	Derivative of the classifier loss w.r.t.
grad_tr_params(self, x, y)	Derivative of the classifier training objective w.r.t. the classifier
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
hessian_tr_params(self, x[, y])	Hessian of the training objective w.r.t.
is_fitted(self)	Return True if the classifier is trained (fitted).
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
predict(self, x[, return_decision_function])	Perform classification of each pattern in x.
save(self, path)	Save class object to file.

Continued on next page

Table 75 – continued from previous page

save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property C
Constant that multiplies the regularization term.

Equal to 1 / alpha.

property b
Bias term.

property w
Vector with feature weights (dense or sparse).

CClassifierSGD

class secml.ml.classifiers.sklearn.c_classifier_sgd.**CClassifierSGD**(*loss*, *regularizer*, *alpha=0.01*, *fit_intercept=True*, *max_iter=1000*, *tol=None*, *shuffle=True*, *learning_rate='optimal'*, *eta0=10.0*, *power_t=0.5*, *class_weight=None*, *warm_start=False*, *average=False*, *random_state=None*, *preprocess=None*)

Bases: *secml.ml.classifiers.c_classifier_linear.CClassifierLinearMixin*, *secml.ml.classifiers.sklearn.c_classifier_sklearn.CClassifierSkLearn*, secml.ml.classifiers.gradients.mixin_classifier_gradient_sgd.CClassifierGradientSGDMixin

Stochastic Gradient Descent Classifier.

Parameters

loss [CLoss] Loss function to be used during classifier training.

regularizer [CRegularizer] Regularizer function to be used during classifier training.

kernel [None or CKernel subclass, optional] Deprecated since version 0.12.

Instance of a CKernel subclass to be used for computing similarity between patterns. If None (default), a linear SVM will be created. In the future this parameter will be removed from this classifier and kernels will have to be passed as preprocess.

alpha [float, optional] Constant that multiplies the regularization term. Default 0.01. Also used to compute learning_rate when set to 'optimal'.

fit_intercept [bool, optional] If True (default), the intercept is calculated, else no intercept will be used in calculations (e.g. data is expected to be already centered).

max_iter [int, optional] The maximum number of passes over the training data (aka epochs). Default 1000.

tol [float or None, optional] The stopping criterion. If it is not None, the iterations will stop when (loss > best_loss - tol) for 5 consecutive epochs. Default None.

shuffle [bool, optional] If True (default) the training data is shuffled after each epoch.

learning_rate [str, optional] The learning rate schedule. If 'constant', eta = eta0; if 'optimal' (default), eta = 1.0 / (alpha * (t + t0)), where t0 is chosen by a heuristic proposed by Leon Bottou; if 'invscaling', eta = eta0 / pow(t, power_t); if 'adaptive', eta = eta0, as long as the training keeps decreasing.

eta0 [float, optional] The initial learning rate for the 'constant', 'invscaling' or 'adaptive' schedules. Default 10.0.

power_t [float, optional] The exponent for inverse scaling learning rate. Default 0.5.

class_weight [{dict, 'balanced', None}, optional] Set the parameter C of class i to *class_weight[i] * C*. If not given (default), all classes are supposed to have weight one. The 'balanced' mode uses the values of labels to automatically adjust weights inversely proportional to class frequencies as *n_samples / (n_classes * np.bincount(y))*.

warm_start [bool, optional] If True, reuse the solution of the previous call to fit as initialization, otherwise, just erase the previous solution. Default False.

average [bool or int, optional] If True, computes the averaged SGD weights and stores the result in the *coef_* attribute. If set to an int greater than 1, averaging will begin once the total number of samples seen reaches average. Default False.

random_state [int, RandomState or None, optional] The seed of the pseudo random number generator to use when shuffling the data. If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*. Default None.

preprocess [CPreProcess or str or None, optional] Features preprocess to be applied to input data. Can be a CPreProcess subclass or a string with the type of the desired preprocessor. If None, input data is used as is.

Attributes

`class_type` ['sgd'] Defines class type.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
decision_function(self, x[, y])	Computes the decision function for each pattern in x.
deepcopy(self)	Returns a deep copy of current class.
estimate_parameters(self, dataset, ... [, ...])	Estimate parameter that give better result respect a chose metric.
fit(self, x, y)	Trains the classifier.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class and SkLearn model parameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
grad_f_params(self, x[, y])	Derivative of the decision function w.r.t.
grad_f_x(self, x, y)	Computes the gradient of the classifier's decision function wrt x.
grad_loss_params(self, x, y[, loss])	Derivative of the classifier loss w.r.t.
grad_tr_params(self, x, y)	Derivative of the classifier training objective function w.r.t.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
hessian_tr_params(self, x, y)	Hessian of the training objective w.r.t.
is_fitted(self)	Return True if the classifier is trained (fitted).
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
predict(self, x[, return_decision_function])	Perform classification of each pattern in x.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property C
Constant that multiplies the regularization term.

Equal to 1 / alpha.

property b
Bias term.

property loss
Returns the loss function used by classifier.

property regularizer

Returns the regularizer function used by classifier.

property w
Vector with feature weights (dense or sparse).

CClassifierSVM

class secml.ml.classifiers.sklearn.c_classifier_svm.**CClassifierSVM**(*C=1.0*, *kernel=None*, *class_weight=None*, *preprocess=None*, *n_jobs=1*)

Bases: *secml.ml.classifiers.c_classifier.CClassifier*

Support Vector Machine (SVM) classifier.

Parameters

C [float, optional] Penalty hyper-parameter C of the error term. Default 1.0.

kernel [None or CKernel subclass, optional] Instance of a CKernel subclass to be used for computing similarity between patterns. If None (default), a linear SVM is trained in the primal; otherwise an SVM is trained in the dual, using the precomputed kernel values.

class_weight [{dict, 'balanced', None}, optional] Set the parameter C of class i to *class_weight[i] * C*. If not given (default), all classes are supposed to have weight one. The 'balanced' mode uses the values of labels to automatically adjust weights inversely proportional to class frequencies as *n_samples / (n_classes * np.bincount(y))*.

preprocess [CModule or str or None, optional] Features preprocess to be applied to input data. Can be a CPreProcess subclass or a string with the type of the desired preprocessor. If None, input data is used as is.

n_jobs [int, optional] Number of parallel workers to use for the classifier. Cannot be higher than processor's number of cores. Default is 1.

See also:

CKernel Pairwise kernels and metrics.

Notes

Current implementation relies on `sklearn.svm.SVC` for the training step.

Attributes

class_type ['svm'] Defines class type.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
decision_function(self, x[, y])	Computes the decision function for each pattern in x.
deepcopy(self)	Returns a deep copy of current class.
estimate_parameters(self, dataset, ... [, ...])	Estimate parameter that give better result respect a chose metric.
fit(self, x, y)	Trains the classifier.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
grad_f_params(self, x[, y])	Derivative of the decision function w.r.t.
grad_f_x(self, x, y)	Computes the gradient of the classifier's decision function wrt x.
grad_loss_params(self, x, y[, loss])	Derivative of the loss w.r.t.
grad_tr_params(self, x, y)	Derivative of the classifier training objective w.r.t.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
hessian_tr_params(self[, x, y])	Hessian of the training objective w.r.t.
is_fitted(self)	Return True if the classifier is trained (fitted).
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
predict(self, x[, return_decision_function])	Perform classification of each pattern in x.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property C
Penalty parameter C of the error term.

property alpha
Signed coefficients of the SVs in the decision function.

property b

property class_weight
Weight of each training class.

grad_f_params (*self*, *x*, *y=1*)
Derivative of the decision function w.r.t. alpha and b

Parameters

x [CArray] Samples on which the training objective is computed.

y [int] Index of the class wrt the gradient must be computed.

grad_loss_params (*self*, *x*, *y*, *loss=None*)
Derivative of the loss w.r.t. the classifier parameters (alpha, b)

dL / d_params = dL / df * df / d_params

Parameters

x [CArray] Features of the dataset on which the loss is computed.

y [CArray] Labels of the training samples.

loss: None (default) or CLoss If the loss is equal to None (default) the classifier loss is used to compute the derivative.

grad_tr_params (*self*, *x*, *y*)
Derivative of the classifier training objective w.r.t. the classifier parameters.

dL / d_params = dL / df * df / d_params + dReg / d_params

Parameters

x [CArray] Features of the dataset on which the loss is computed.

y [CArray] Features of the training samples

hessian_tr_params (*self*, *x=None*, *y=None*)
Hessian of the training objective w.r.t. the classifier parameters.

property kernel
Kernel type (None or string).

property sv_idx
Indices of Support Vectors within the training dataset.

property w

CClassifierDNN

class secml.ml.classifiers.c_classifier_dnn.**CClassifierDNN**(*model*, *input_shape=None*, *preprocess=None*, *pretrained=False*, *pretrained_classes=None*, *softmax_outputs=False*, *n_jobs=1*)

Bases: *secml.ml.classifiers.c_classifier.CClassifier*

CClassifierDNN, wrapper for DNN models.

Parameters

model [model dtype of the specific backend] The model to wrap.

input_shape [tuple or None, optional] Shape of the input for the DNN, it will be used for reshaping the input data to the expected shape.

preprocess [CPreprocess or str or None, optional] Preprocessing module.

pretrained [bool, optional] Whether or not the model is pretrained. If the model is pretrained, the user won't need to call *fit* after loading the model. Default False.

pretrained_classes [None or CArray, optional] List of classes labels if the model is pretrained. If set to None, the class labels for the pretrained model should be inferred at the moment of initialization of the model and set to CArray.arange(n_classes). Default None.

softmax_outputs [bool, optional] Whether or not to add a softmax layer after the logits. Default False.

n_jobs [int, optional] Number of parallel workers to use for training the classifier. Cannot be higher than processor's number of cores. Default is 1.

Attributes

class_type ['dnn-clf'] Defines class type.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
decision_function(self, x[, y])	Computes the decision function for each pattern in x.
deepcopy(self)	Returns a deep copy of current class.
estimate_parameters(self, dataset, ..., [, ...])	Estimate parameter that give better result respect a chose metric.
fit(self, x, y)	Trains the classifier.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_layer_gradient(self, x, w[, layer])	Computes the gradient of the classifier's decision function wrt input.
get_layer_output(self, x[, layer])	Returns the output of the desired net layer(s).
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
grad_f_x(self, x, y)	Computes the gradient of the classifier's decision function wrt x.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
is_fitted(self)	Return True if the classifier is trained (fitted).
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_model(self, filename)	Restores the model and optimization parameters.
load_state(self, path)	Sets the object state from file.
predict(self, x[, return_decision_function])	Perform classification of each pattern in x.
save(self, path)	Save class object to file.
save_model(self, filename)	Stores the model and optimization parameters.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.

Continued on next page

Table 78 – continued from previous page

set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

get_layer_gradient (*self*, *x*, *w*, *layer=None*)
Computes the gradient of the classifier's decision function wrt input.

Parameters

x [CArray] Input sample

w [CArray] Will be passed to backward and must have a proper shape depending on the chosen output layer (the last one if *layer* is None). This is required if *layer* is not None.

layer [str or None, optional] Name of the layer. If None, the gradient at the last layer will be returned and *y* is required if *w* is None or softmax_outputs is True. If not None, *w* of proper shape is required.

Returns

gradient [CArray] Gradient of the classifier's df wrt its input. Vector-like array.

get_layer_output (*self*, *x*, *layer=None*)
Returns the output of the desired net layer(s).

Parameters

x [CArray] Input data.

layer [str or None, optional] Name of the layer to get the output from. If None, the output of the last layer will be returned.

Returns

CArray Output of the desired layer.

property input_shape
Returns the input shape of the first layer of the neural network.

property layer_names
Returns the names of the layers of the model.

abstract property layer_shapes
Returns a dictionary containing the shapes of the output of each layer of the model.

abstract property layers
Returns list of tuples containing the layers of the model. Each tuple is structured as (layer_name, layer).

abstract load_model (*self*, *filename*)
Restores the model and optimization parameters. Notes: the model class should be defined before loading the params.

Parameters

filename [str] path where to find the stored model

abstract save_model (*self*, *filename*)
Stores the model and optimization parameters.

Parameters

filename [str] path of the file for storing the model

property softmax_outputs

CClassifierPyTorch

class secml.ml.classifiers.pytorch.c_classifier_pytorch.**CClassifierPyTorch**(*model*, *loss=None*, *optimizer=None*, *optimizer_scheduler=None*, *pretrained=False*, *pretrained_classes=None*, *input_shape=None*, *random_state=None*, *preprocess=None*, *softmax_outputs=False*, *epochs=10*, *batch_size=1*, *n_jobs=1*)

Bases: *secml.ml.classifiers.c_classifier_dnn.CClassifierDNN*, secml.ml.classifiers.gradients.mixin_classifier_gradient.CClassifierGradientMixin

CClassifierPyTorch, wrapper for PyTorch models.

Parameters

model: *torch.nn.Module* object to use as classifier

loss: loss object from *torch.nn*

optimizer: optimizer object from *torch.optim*

random_state: int or None, optional random state to use for initializing the model weights. Default value is None.

preprocess: preprocessing module.

softmax_outputs: bool, optional if set to True, a softmax function will be applied to the return value of the decision function. Note: some implementation adds the softmax function to the network class as last layer or last forward function, or even in the loss function (see torch.nn.CrossEntropyLoss). Be aware that the softmax may have already been applied. Default value is False.

epochs: int number of epochs for training the neural network. Default value is 10.

batch_size: int size of the batches to use for loading the data. Default value is 1.

n_jobs: int number of workers to use for data loading and processing. This parameter follows the library expected behavior of having 1 worker as the main process. The loader will spawn *n_jobs-1* workers. Default value for n_jobs is 1 (zero additional workers spawned).

Attributes

class_type ['pytorch-clf'] Defines class type.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
decision_function(self, x[, y])	Computes the decision function for each pattern in x.
deepcopy(self)	Returns a deep copy of current class.
estimate_parameters(self, dataset, ..., [, ...])	Estimate parameter that give better result respect a chose metric.
fit(self, x, y)	Trains the classifier.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_layer_gradient(self, x, w[, layer])	Computes the gradient of the classifier's decision function wrt input.
get_layer_output(self, x[, layer])	Returns the output of the desired net layer(s).
get_params(self)	Returns the dictionary of class parameters.
get_state(self[, return_optimizer])	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
grad_f_params(self, x, y)	Derivative of the decision function w.r.t.
grad_f_x(self, x, y)	Computes the gradient of the classifier's decision function wrt x.
grad_loss_params(self, x, y[, loss])	Derivative of a given loss w.r.t.
grad_tr_params(self, x, y)	Derivative of the classifier training objective function w.r.t.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
hessian_tr_params(self, x, y)	Hessian of the training objective w.r.t.
hook_layer_output(self[, layer_names])	Creates handlers for the hooks that store the layer outputs.
is_fitted(self)	Return True if the classifier is trained (fitted).
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_model(self, filename[, classes])	Restores the model and optimizer's parameters.
load_state(self, path)	Sets the object state from file.
predict(self, x[, return_decision_function])	Perform classification of each pattern in x.
save(self, path)	Save class object to file.
save_model(self, filename)	Stores the model and optimizer's parameters.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

get_layer_shape	

property batch_size
Returns the batch size used for the dataset loader.

property epochs
Returns the number of epochs for which the model will be trained.

get_layer_shape(*self*, *layer_name*)

get_params(*self*)
Returns the dictionary of class parameters.

get_state(*self*, *return_optimizer=True*)
Returns the object state dictionary.

Parameters

return_optimizer [bool, optional] If True (default), state of *optimizer* and *optimizer_scheduler*, if defined, will be included in the state dictionary.

Returns

dict Dictionary containing the state of the object.

hook_layer_output(*self*, *layer_names=None*)
Creates handlers for the hooks that store the layer outputs.

Parameters

layer_names [list or str, optional] List of layer names to hook. Cleans previously defined hooks to prevent multiple hook creations.

property layer_shapes
Returns a dictionary containing the shapes of the output of each layer of the model.

property layers
Returns the layers of the model, if possible.

load_model(*self*, *filename*, *classes=None*)
Restores the model and optimizer's parameters. Notes: the model class and optimizer should be defined before loading the params.

Parameters

filename [str] path where to find the stored model

classes [list, tuple or None, optional] This parameter is used only if the model was stored with native PyTorch. Class labels (sorted) for matching classes to indexes in the loaded model. If classes is None, the classes will be assigned new indexes from 0 to n_classes.

property loss
Returns the loss function used by classifier.

property model
Returns the model used by classifier.

property optimizer
Returns the optimizer used by classifier.

property optimizer_scheduler
Returns the optimizer used by classifier.

save_model(*self*, *filename*)
Stores the model and optimizer's parameters.

Parameters

filename [str] path of the file for storing the model

set_state(*self*, *state_dict*, *copy=False*)
Sets the object state using input dictionary.

property trained
True if the model has been trained.

secml.ml.classifiers.pytorch.c_classifier_pytorch.**get_layers**(*net*)

clf_utils

secml.ml.classifiers.clf_utils.**check_binary_labels**(*labels*)
Check if input labels are binary {0, +1}.

Parameters

labels [CArray or int] Binary labels to be converted. As of PRALib convention, binary labels are {0, +1}.

Raises

ValueError If input labels are not binary.

secml.ml.classifiers.clf_utils.**convert_binary_labels**(*labels*)
Convert input binary labels to {-1, +1}.

Parameters

labels [CArray or int] Binary labels in {0, +1} to be converted to {-1, +1}.

Returns

converted_labels [CArray or int] Binary labels converted to {-1, +1}.

Examples

```
>>> from secml.ml.classifiers.clf_utils import convert_binary_labels
>>> from secml.array import CArray
```

```
>>> print(convert_binary_labels(0))
-1
```

```
>>> print(convert_binary_labels(CArray([0,1,1,1,0,0])))
CArray([-1  1  1  1 -1 -1])
```

secml.ml.features

secml.ml.features.normalization

CNormalizer

class secml.ml.features.normalization.c_normalizer.**CNormalizer**(*preprocess=None*)

Bases: *secml.ml.features.c_preprocess.CPreProcess*

Common interface for normalization preprocessing algorithms.

Attributes

class_type Defines class type.

logger Logger for current object.

n_jobs

preprocess Inner preprocessor (if any).

verbose Verbosity level of logger output.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
deepcopy(self)	Returns a deep copy of current class.
fit(self, x[, y])	Fit the preprocessor.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
fit_transform(self, x[, y])	Fit preprocessor using data and then transform data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
inverse_transform(self, x)	Revert data to original form.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

Continued on next page

Table 80 – continued from previous page

transform(self, x)	Apply the transformation algorithm on data.

CNormalizerLinear

class secml.ml.features.normalization.c_normalizer_linear.**CNormalizerLinear**(*preprocess=None*)

Bases: *secml.ml.features.normalization.c_normalizer.CNormalizer*

Standardizes array by linearly scaling each feature.

Input data must have one row for each patterns, so features to scale are on each array's column.

The standardization is given by:

```
X_scaled = m * X(axis=0) + q
```

where m, q are specific constants for each normalization.

Notes

Differently from numpy, we manage flat vectors as 2-Dimensional of shape (1, array.size). This means that normalizing a flat vector is equivalent to transform array.atleast_2d(). To obtain a numpy-style normalization of flat vectors, transpose array first.

Attributes

b Returns the bias of the linear normalizer.

class_type Defines class type.

logger Logger for current object.

n_jobs

preprocess Inner preprocessor (if any).

verbose Verbosity level of logger output.

w Returns the step of the linear normalizer.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
deepcopy(self)	Returns a deep copy of current class.
fit(self, x[, y])	Fit the preprocessor.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
fit_transform(self, x[, y])	Fit preprocessor using data and then transform data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.

Continued on next page

Table 81 – continued from previous page

get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
inverse_transform(self, x)	Revert data to original form.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.
transform(self, x)	Apply the transformation algorithm on data.

abstract property b
Returns the bias of the linear normalizer.

abstract property w
Returns the step of the linear normalizer.

CNormalizerMeanStd

class secml.ml.features.normalization.c_normalizer_mean_std.**CNormalizerMeanStd**(*mean=None*, *std=None*, *with_std=True*, *preprocess=None*)

Bases: *secml.ml.features.normalization.c_normalizer_linear.CNormalizerLinear*

Normalize with given mean and standard deviation.

If mean/std are tuples of multiple values, input is expected to be uniformly splittable in a number of channels equal to the number of values in the tuples. Both input tuples must have the same length.

Result will be: (input[channel] - mean[channel]) / std[channel]

If mean and std are None, values to use as mean and std will be computed from data. The result wil be an array with 0 mean or/and unit variance (if with_std parameter is True, default). In this case, the standard deviation calculated by numpy is the maximum likelihood estimate, i.e. the second moment of the set of values about their mean. See also *CArray.std* for more information.

Parameters

mean [scalar or tuple of scalars or None, optional] Mean to use for normalization. If a tuple, each value represent a channel of the input. The number of features of the training data should be divisible by the number of values of the tuple. If a scalar, the same value is applied to all features. If None, mean is computed from training data. Cannot be None if *std* is not None and *with_std* is True.

std [scalar or tuple of scalars or None, optional] Variance to use for normalization. If a tuple, each value represent a channel of the input. The number of features of the training data should be divisible by the number of values of the tuple. If a scalar, the same value is

applied to all features. If None, std is computed from training data. Cannot be None if *mean* is not None and *with_std* is True.

with_std [bool, optional] If True (default), normalizer scales array using std too. If False, *std* parameter is ignored.

preprocess [CPreProcess or str or None, optional] Features preprocess to be applied to input data. Can be a CPreProcess subclass or a string with the type of the desired preprocessor. If None, input data is used as is.

Attributes

class_type ['mean-std'] Defines class type.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
deepcopy(self)	Returns a deep copy of current class.
fit(self, x[, y])	Fit the preprocessor.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
fit_transform(self, x[, y])	Fit preprocessor using data and then transform data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
inverse_transform(self, x)	Revert data to original form.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.
transform(self, x)	Apply the transformation algorithm on data.

property b

Returns the bias of the linear normalizer.

property mean

Mean to use for normalization.

One value for each training array feature.

property std

Variance to use for normalization.

One value for each training array feature.

property w
Returns the slope of the linear normalizer.

property with_std
True if normalizer should transform array using variance too.

CNormalizerMinMax

class secml.ml.features.normalization.c_normalizer_minmax.**CNormalizerMinMax**(*feature_range=None*, *preprocess=None*)

Bases: *secml.ml.features.normalization.c_normalizer_linear.CNormalizerLinear*

Standardizes array by scaling each feature to a given range.

This estimator scales and translates each feature individually such that it is in the given range on the training array, i.e. between zero and one.

Input data must have one row for each patterns, so features to scale are on each array's column.

The standardization is given by:

```
X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
X_scaled = X_std * (max - min) + min
```

where min, max = feature_range.

Parameters

feature_range [tuple of scalars or None, optional] Desired range of transformed data, tuple of 2 scalars where *feature_range[0]* is the minimum and *feature_range[1]* is the maximum value. If feature_range is None, features will be scaled using (0., 1.) range.

preprocess [CPreProcess or str or None, optional] Features preprocess to be applied to input data. Can be a CPreProcess subclass or a string with the type of the desired preprocessor. If None, input data is used as is.

Notes

Differently from numpy, we manage flat vectors as 2-Dimensional of shape (1, array.size). This means that normalizing a flat vector is equivalent to transform array.atleast_2d(). To obtain a numpy-style normalization of flat vectors, transpose array first.

Examples

```
>>> from secml.array import CArray
>>> from secml.ml.features.normalization import CNormalizerMinMax
>>> array = CArray([[1., -1., 2.], [2., 0., 0.], [0., 1., -1.]])
```

```
>>> print(CNormalizerMinMax().fit_transform(array))
CArray([[0.5      0.       1.      ]
 [1.       0.5      0.333333]
 [0.       1.       0.      ]])
```

```
>>> print(CNormalizerMinMax(feature_range=(-1,1)).fit_transform(array))
CArray([[ 0.       -1.        1.      ]
 [ 1.        0.       -0.333333]
 [-1.        1.       -1.      ]])
```

Attributes

class_type ['min-max'] Defines class type.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
deepcopy(self)	Returns a deep copy of current class.
fit(self, x[, y])	Fit the preprocessor.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
fit_transform(self, x[, y])	Fit preprocessor using data and then transform data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
inverse_transform(self, x)	Revert data to original form.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.
transform(self, x)	Apply the transformation algorithm on data.

property b
Returns the bias of the linear normalizer.

property feature_range
Desired range of transformed data.

property m
Returns the slope of the linear normalizer. (excluding the feature range).

property max
Maximum of training array per feature.

Returns

train_max [CArray] Flat dense array with the maximum of each feature of the training array. If the scaler has not been trained yet, returns None.

property min
Minimum of training array per feature.

Returns

train_min [CArray] Flat dense array with the minimum of each feature of the training array. If the scaler has not been trained yet, returns None.

property q
Returns the bias of the linear normalizer (excluding the feature range).

property w
Returns the slope of the linear normalizer.

CNormalizerUnitNorm

class secml.ml.features.normalization.c_normalizer_unitnorm.**CNormalizerUnitNorm**(*norm='l2'*, *preprocess=None*)

Bases: *secml.ml.features.normalization.c_normalizer.CNormalizer*

Normalize patterns individually to unit norm.

Each pattern (i.e. each row of the data matrix) with at least one non zero component is rescaled independently of other patterns so that its norm (l1 or l2 or max) equals one.

For the Row normalizer, no training routine is needed, so using fit_normalize() method is suggested for clarity. Use fit() method, which does nothing, only to streamline a pipelined environment.

Parameters

norm [{'l1', 'l2', 'max'}, optional] Order of the norm to normalize each pattern with.'l2' is the default.

preprocess [CPreProcess or str or None, optional] Features preprocess to be applied to input data. Can be a CPreProcess subclass or a string with the type of the desired preprocessor. If None, input data is used as is.

Notes

Differently from numpy, we manage flat vectors as 2-Dimensional of shape (1, array.size). This means that normalizing a flat vector is equivalent to transform array.atleast_2d(). To obtain a numpy-style normalization of flat vectors, transpose array first.

Examples

```
>>> from secml.array import CArray
>>> from secml.ml.features.normalization import CNormalizerUnitNorm
>>> array = CArray([[1., -1., 2.], [2., 0., 0.], [0., 1., -1.]])
```

```
>>> dense_normalized = CNormalizerUnitNorm(norm="l2").fit_transform(array)
>>> print(dense_normalized)
CArray([[ 0.408248 -0.408248  0.816497]
 [ 1.        0.        0.      ]
 [ 0.        0.707107 -0.707107]])
```

```
>>> dense_normalized =(CNormalizerUnitNorm(norm="l1").fit_transform(array))
>>> print(dense_normalized)
CArray([[ 0.25 -0.25  0.5 ]
 [ 1.    0.    0.  ]
 [ 0.    0.5  -0.5 ]])
```

```
>>> print(array / array.norm_2d(order=1, axis=1, keepdims=True))
CArray([[ 0.25 -0.25  0.5 ]
 [ 1.    0.    0.  ]
 [ 0.    0.5  -0.5 ]])
```

Attributes

class_type [‘unit-norm’] Defines class type.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
deepcopy(self)	Returns a deep copy of current class.
fit(self, x[, y])	Fit the preprocessor.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
fit_transform(self, x[, y])	Fit preprocessor using data and then transform data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.

Continued on next page

Table 84 – continued from previous page

inverse_transform(self, x)	Revert data to original form.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.
transform(self, x)	Apply the transformation algorithm on data.

property norm
Return the norm of each training array's patterns.

CNormalizerDNN

class secml.ml.features.normalization.c_normalizer_dnn.**CNormalizerDNN**(*net*, *out_layer=None*, *preprocess=<no value>*)

Bases: *secml.ml.features.normalization.c_normalizer.CNormalizer*

Normalized features are the DNN deepfeatures

Parameters

net [CClassifierDNN] DNN to be used for extracting deepfeatures. This must be already trained.

out_layer [str or None, optional] Identifier of the layer at which the features must be retrieved. If None, the output of last layer will be returned.

Notes

Any additional inner preprocess should not be passed as the *preprocess* parameter but to the DNN instead.

Attributes

class_type ['dnn'] Defines class type.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
deepcopy(self)	Returns a deep copy of current class.
fit(self, x, y)	Fit normalization algorithm using data.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
fit_transform(self, x[, y])	Fit preprocessor using data and then transform data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
inverse_transform(self, x)	Revert data to original form.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.
transform(self, x)	Apply the transformation algorithm on data.

fit (*self*, *x*, *y*)

Fit normalization algorithm using data.

This fit function is just a placeholder and simply returns the normalizer itself.

Parameters

x [CArray] Array to be used for training normalization algorithm. Shape of input array depends on the algorithm itself.

y [CArray or None, optional] Flat array with the label of each pattern. Not Used.

Returns

CNormalizer Instance of the trained normalizer.

property net

The DNN.

secml.ml.features.reduction

CReducer

class secml.ml.features.reduction.c_reducer.**CReducer**(*preprocess=None*)

Bases: *secml.ml.features.c_preprocess.CPreProcess*

Interface for feature dimensionality reduction algorithms.

Attributes

class_type Defines class type.

logger Logger for current object.

n_jobs

preprocess Inner preprocessor (if any).

verbose Verbosity level of logger output.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
deepcopy(self)	Returns a deep copy of current class.
fit(self, x[, y])	Fit the preprocessor.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
fit_transform(self, x[, y])	Fit preprocessor using data and then transform data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
inverse_transform(self, x)	Revert data to original form.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.
transform(self, x)	Apply the transformation algorithm on data.

CLDA

class secml.ml.features.reduction.c_reducer_lda.**CLDA**(*n_components=None*, *preprocess=None*)

Bases: *secml.ml.features.reduction.c_reducer.CReducer*

Linear Discriminant Analysis (LDA).

Parameters

preprocess [CPreProcess or str or None, optional] Features preprocess to be applied to input data. Can be a CPreProcess subclass or a string with the type of the desired preprocessor. If None, input data is used as is.

Attributes

class_type ['lda'] Defines class type.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
deepcopy(self)	Returns a deep copy of current class.
fit(self, x[, y])	Fit the preprocessor.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
fit_transform(self, x[, y])	Fit preprocessor using data and then transform data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
inverse_transform(self, x)	Revert data to original form.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.
transform(self, x)	Apply the transformation algorithm on data.

property classes

Unique targets used for training.

property eigenvec

Eigenvectors estimated from the training data. Is a matrix of shape: n_eigenvectors * n_features.

property lda
Trained sklearn LDA transformer.

property mean
Per-feature empirical mean, estimated from the training data.

CPCA

class secml.ml.features.reduction.c_reducer_pca.**CPCA**(*n_components=None*, *preprocess=None*)

Bases: *secml.ml.features.reduction.c_reducer.CReducer*

Principal Component Analysis (PCA).

Parameters

preprocess [CPreProcess or str or None, optional] Features preprocess to be applied to input data. Can be a CPreProcess subclass or a string with the type of the desired preprocessor. If None, input data is used as is.

Attributes

class_type ['pca'] Defines class type.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
deepcopy(self)	Returns a deep copy of current class.
fit(self, x[, y])	Fit the preprocessor.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
fit_transform(self, x[, y])	Fit preprocessor using data and then transform data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
inverse_transform(self, x)	Revert data to original form.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.

Continued on next page

Table 88 – continued from previous page

set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.
transform(self, x)	Apply the transformation algorithm on data.

property components
Eigenvectors of inverse training array.

property eigenval
Eigenvalues estimated from the training data.

property eigenvec
Eigenvectors estimated from the training data.

property explained_variance
Variance explained by each of the selected components.

property explained_variance_ratio
Percentage of variance explained by each of the selected components.

If n_components is None, then all components are stored and the sum of explained variances is equal to 1.0

property mean
Per-feature empirical mean, estimated from the training data.

CPreProcess

class secml.ml.features.c_preprocess.**CPreProcess**(*preprocess=None*)
Bases: secml.ml.c_module.CModule

Common interface for feature preprocessing algorithms.

Parameters

preprocess [CPreProcess or str or None, optional] Features preprocess to be applied to input data. Can be a CPreProcess subclass or a string with the type of the desired preprocessor. If None, input data is used as is.

Attributes

class_type Defines class type.

logger Logger for current object.

n_jobs

preprocess Inner preprocessor (if any).

verbose Verbosity level of logger output.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
deepcopy(self)	Returns a deep copy of current class.
fit(self, x[, y])	Fit the preprocessor.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
fit_transform(self, x[, y])	Fit preprocessor using data and then transform data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
inverse_transform(self, x)	Revert data to original form.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.
transform(self, x)	Apply the transformation algorithm on data.

fit (*self*, *x*, *y=None*)
Fit the preprocessor.

Parameters

x [CArray] Array to be used as training set. Each row must correspond to one single patterns, so each column is a different feature.

y [CArray or None, optional] Flat array with the label of each pattern. Can be None if not required by the preprocessing algorithm.

Returns

CPreProcess Instance of the trained normalizer.

fit_transform (*self*, *x*, *y=None*)
Fit preprocessor using data and then transform data.

This method is equivalent to call fit(data) and transform(data) in sequence, but it's useful when data is both the training array and the array to be transformed.

Parameters

x [CArray] Array to be transformed. Each row must correspond to one single patterns, so each column is a different feature.

y [CArray or None, optional] Flat array with the label of each pattern. Can be None if not required by the preprocessing algorithm.

Returns

CArray Transformed input data.

See also:

fit fit the preprocessor.

transform transform input data.

inverse_transform(*self*, *x*)

Revert data to original form.

Parameters

x [CArray] Transformed array to be reverted to original form. Shape of input array depends on the algorithm itself.

Returns

CArray Original input data.

Warning: Reverting a transformed array is not always possible. See description of each preprocessor for details.

transform(*self*, *x*)

Apply the transformation algorithm on data.

Parameters

x [CArray] Array to be transformed. Shape of input array depends on the algorithm itself.

Returns

CArray Transformed input data.

secml.ml.kernels

CKernel

class secml.ml.kernels.c_kernel.**CKernel**(*preprocess=None*)

Bases: secml.ml.c_module.CModule

Abstract class that defines basic methods for kernels.

A kernel is a pairwise metric that compute the distance between sets of patterns.

Kernels can be considered similarity measures, i.e. s(a, b) > s(a, c) if objects a and b are considered "more similar" than objects a and c. A kernel must be positive semi-definite (PSD), even though non-PSD kernels can also be used to train classifiers (e.g., SVMs, but losing convexity).

Parameters

preprocess [CModule or None, optional] Features preprocess to be applied to input data. Can be a CModule subclass. If None, input data is used as is.

Attributes

class_type Defines class type.

logger Logger for current object.

n_jobs

preprocess Inner preprocessor (if any).

rv Reference vectors with respect to compute the kernel.

verbose Verbosity level of logger output.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
deepcopy(self)	Returns a deep copy of current class.
fit(self, x, y)	Fit estimator.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
k(self, x[, rv])	Compute kernel between x and rv.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

k (*self*, *x*, *rv=None*)

Compute kernel between x and rv.

Parameters

x [CArray] First array of shape (n_x, n_features).

rv [CArray, optional] Second array of shape (n_rv, n_features). If not specified, it is set to x and the kernel k(x,x) is computed.

Returns

kernel [CArray or scalar] Kernel between x and rv. Array of shape (n_x, n_rv) or scalar if both x and y are vector-like.

Examples

```
>>> from secml.array import CArray
>>> from secml.ml.kernels import CKernelRBF
```

```
>>> array1 = CArray([[15,25],[45,55]])
>>> array2 = CArray([[10,20],[40,50]])
>>> print(CKernelRBF().k(array1, array2))
CArray([[1.92875e-22 0.00000e+00]
 [0.00000e+00 1.92875e-22]])
```

```
>>> print(CKernelRBF().k(array1))
CArray([[1. 0.]
 [0. 1.]])
```

```
>>> vector = CArray([15,25])
>>> print(CKernelRBF().k(vector, array1))
CArray([[1. 0.]])
>>> print(CKernelRBF().k(array1, vector))
CArray([[1.]
 [0.]])
>>> print(CKernelRBF().k(vector, vector))
CArray([[1.]])
```

property rv
Reference vectors with respect to compute the kernel.

CKernelChebyshevDistance

class secml.ml.kernels.c_kernel_chebyshev_distance.**CKernelChebyshevDistance**(*preprocess=None*)
Bases: *secml.ml.kernels.c_kernel.CKernel*

Chebyshev distance kernel.

Given matrices X and RV, this is computed as:

```
K(x, rv) = max(|x - rv|)
```

for each pair of rows in X and in RV.

Examples

```
>>> from secml.array import CArray
>>> from secml.ml.kernels import CKernelChebyshevDistance
```

```
>>> x = CArray([[1,2],[3,4]])
>>> v = CArray([[5,6],[7,8]])
>>> print(CKernelChebyshevDistance().k(x,v))
CArray([[-4. -6.]
 [-2. -4.]])
```

```
>>> print(CKernelChebyshevDistance().k(x))
CArray([[-0. -2.]
 [-2. -0.]])
```

Attributes

class_type ['chebyshev-dist'] Defines class type.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
deepcopy(self)	Returns a deep copy of current class.
fit(self, x, y)	Fit estimator.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
k(self, x[, rv])	Compute kernel between x and rv.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

CKernelEuclidean

class secml.ml.kernels.c_kernel_euclidean.**CKernelEuclidean**(*squared=False*, *preprocess=None*)

Bases: *secml.ml.kernels.c_kernel.CKernel*

Euclidean distance kernel.

Given matrices X and RV, this is computed as the negative Euclidean dist.:

```
K(x, rv) = -sqrt(dot(x, x) - 2 * dot(x, rv) + dot(rv, rv))
```

for each pair of rows in X and in RV. If parameter squared is True (default False), sqrt() operation is avoided.

Parameters

squared [bool, optional] If True, return squared Euclidean distances. Default False.

preprocess [CModule or None, optional] Features preprocess to be applied to input data. Can be a CModule subclass. If None, input data is used as is.

Examples

```
>>> from secml.array import CArray
>>> from secml.ml.kernels.c_kernel_euclidean import CKernelEuclidean
```

```
>>> print(CKernelEuclidean().k(CArray([[1,2],[3,4]]), CArray([[10,20],[30,40]])))
CArray([[-20.124612 -47.801674]
 [-17.464249 -45.      ]])
```

```
>>> print(CKernelEuclidean().k(CArray([[1,2],[3,4]])))
CArray([[0.       -2.828427]
 [-2.828427 0.      ]])
```

Attributes

class_type ['euclidean'] Defines class type.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
deepcopy(self)	Returns a deep copy of current class.
fit(self, x, y)	Fit estimator.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
k(self, x[, rv])	Compute kernel between x and rv.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property rv_norm_squared
Pre-computed dot-products of vectors in rv (e.g., (rv**2).sum(axis=1)).

property squared
If True, squared Euclidean distances are computed.

property x_norm_squared
Pre-computed dot-products of vectors in x (e.g., (x**2).sum(axis=1)).

CKernelHistIntersect

class secml.ml.kernels.c_kernel_histintersect.**CKernelHistIntersect**(*preprocess=None*)
Bases: *secml.ml.kernels.c_kernel.CKernel*

Histogram Intersection Kernel.

Given matrices X and RV, this is computed by:

```
K(x, rv) = sum_i ( min(x[i], rv[i]) )
```

for each pair of rows in X and in RV.

Examples

```
>>> from secml.array import CArray
>>> from secml.ml.kernels.c_kernel_histintersect import CKernelHistIntersect
```

```
>>> print(CKernelHistIntersect().k(CArray([[1,2],[3,4]]), CArray([[10,20],[30,
↪40]])))
CArray([[3. 3.]
 [7. 7.]])
```

```
>>> print(CKernelHistIntersect().k(CArray([[1,2],[3,4]])))
CArray([[3. 3.]
 [3. 7.]])
```

Attributes

class_type ['hist-intersect'] Defines class type.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
deepcopy(self)	Returns a deep copy of current class.
fit(self, x, y)	Fit estimator.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.

Continued on next page

Table 93 – continued from previous page

gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
k(self, x[, rv])	Compute kernel between x and rv.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

CKernelLaplacian

class secml.ml.kernels.c_kernel_laplacian.**CKernelLaplacian**(*gamma=1.0*, *preprocess=None*)

Bases: *secml.ml.kernels.c_kernel.CKernel*

Laplacian Kernel.

Given matrices X and RV, this is computed by:

```
K(x, rv) = exp(-gamma |x-rv|)
```

for each pair of rows in X and in RV.

Parameters

gamma [float] Default is 1.0.

preprocess [CModule or None, optional] Features preprocess to be applied to input data. Can be a CModule subclass. If None, input data is used as is.

Examples

```
>>> from secml.array import CArray
>>> from secml.ml.kernels.c_kernel_laplacian import CKernelLaplacian
```

```
>>> print(CKernelLaplacian(gamma=0.01).k(CArray([[1,2],[3,4]]), CArray([[10,0],[0,
↪40]])))
CArray([[0.895834 0.677057]
 [0.895834 0.677057]])
```

```
>>> print(CKernelLaplacian().k(CArray([[1,2],[3,4]])))
CArray([[1.       0.018316]
 [0.018316 1.      ]])
```

Attributes

class_type ['laplacian'] Defines class type.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
deepcopy(self)	Returns a deep copy of current class.
fit(self, x, y)	Fit estimator.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
k(self, x[, rv])	Compute kernel between x and rv.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property gamma
Gamma parameter.

CKernelLinear

class secml.ml.kernels.c_kernel_linear.**CKernelLinear**(*preprocess=None*)
Bases: *secml.ml.kernels.c_kernel.CKernel*

Linear kernel.

Given matrices X and RV, this is computed by:

```
K(x, rv) = x * rv^T
```

for each pair of rows in X and in RV.

Examples

```
>>> from secml.array import CArray
>>> from secml.ml.kernels.c_kernel_linear import CKernelLinear
```

```
>>> print(CKernelLinear().k(CArray([[1,2],[3,4]]), CArray([[10,20],[30,40]])))
CArray([[ 50. 110.]
 [110. 250.]])
```

```
>>> print(CKernelLinear().k(CArray([[1,2],[3,4]])))
CArray([[ 5. 11.]
 [11. 25.]])
```

Attributes

class_type ['linear'] Defines class type.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
deepcopy(self)	Returns a deep copy of current class.
fit(self, x, y)	Fit estimator.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
k(self, x[, rv])	Compute kernel between x and rv.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

CKernelPoly

class secml.ml.kernels.c_kernel_poly.**CKernelPoly**(*degree=2*, *gamma=1.0*, *coef0=1.0*, *preprocess=None*)

Bases: *secml.ml.kernels.c_kernel.CKernel*

Polynomial kernel.

Given matrices X and RV, this is computed by:

```
K(x, rv) = (coef0 + gamma * <x, rv>)^degree
```

for each pair of rows in X and in RV.

Parameters

degree [int, optional] Kernel degree. Default 2.

gamma [float, optional] Free parameter to be used for balancing. Default 1.0.

coef0 [float, optional] Free parameter used for trading off the influence of higher-order versus lower-order terms in the kernel. Default 1.0.

preprocess [CModule or None, optional] Features preprocess to be applied to input data. Can be a CModule subclass. If None, input data is used as is.

Examples

```
>>> from secml.array import CArray
>>> from secml.ml.kernels.c_kernel_poly import CKernelPoly
```

```
>>> print(CKernelPoly(degree=3, gamma=0.001, coef0=2).k(CArray([[1,2],[3,4]]),␣
↪CArray([[10,20],[30,40]])))
CArray([[ 8.615125  9.393931]
 [ 9.393931 11.390625]])
```

```
>>> print(CKernelPoly().k(CArray([[1,2],[3,4]])))
CArray([[ 36. 144.]
 [144. 676.]])
```

Attributes

class_type ['poly'] Defines class type.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
deepcopy(self)	Returns a deep copy of current class.
fit(self, x, y)	Fit estimator.

Continued on next page

Table 96 – continued from previous page

fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
k(self, x[, rv])	Compute kernel between x and rv.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property coef0
Coef0 parameter.

property degree
Degree parameter.

property gamma
Gamma parameter.

CKernelRBF

class secml.ml.kernels.c_kernel_rbf.**CKernelRBF**(*gamma=1.0*, *preprocess=None*)
Bases: *secml.ml.kernels.c_kernel.CKernel*

Radial basis function (RBF) kernel.

Given matrices X and RV, this is computed by:

```
K(x, rv) = exp(-gamma ||x-rv||^2)
```

for each pair of rows in X and in RV.

Parameters

gamma [float] Default is 1.0. Equals to *-0.5 * sigma^-2* in the standard formulation of rbf kernel, it is a free parameter to be used for balancing.

preprocess [CModule or None, optional] Features preprocess to be applied to input data. Can be a CModule subclass. If None, input data is used as is.

Examples

```
>>> from secml.array import CArray
>>> from secml.ml.kernels.c_kernel_rbf import CKernelRBF
```

```
>>> print(CKernelRBF(gamma=0.001).k(CArray([[1,2],[3,4]]), CArray([[10,20],[30,
↪40]])))
CArray([[0.666977 0.101774]
 [0.737123 0.131994]])
```

```
>>> print(CKernelRBF().k(CArray([[1,2],[3,4]])))
CArray([[1.000000e+00 3.354626e-04]
 [3.354626e-04 1.000000e+00]])
```

Attributes

class_type ['rbf'] Defines class type.

Methods

backward(self[, w])	Returns the preprocessor gradient wrt data.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
create_chain(class_items, kwargs_list)	Creates a chain of preprocessors.
deepcopy(self)	Returns a deep copy of current class.
fit(self, x, y)	Fit estimator.
fit_forward(self, x[, y, caching])	Fit estimator using data and then execute forward on the data.
forward(self, x[, caching])	Forward pass on input x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
gradient(self, x[, w])	Compute gradient at x by doing a backward pass.
k(self, x[, rv])	Compute kernel between x and rv.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property gamma
Gamma parameter.

secml.ml.peval

secml.ml.peval.metrics

CMetric

class secml.ml.peval.metrics.c_metric.**CMetric**

Bases: *secml.core.c_creator.CCreator*

Performance evaluation metrics.

Utility functions to measure classification performance. Some metrics might require probability estimates of the positive class, confidence values, or binary decisions values.

Each metric can be use either y_true (true ground labels) or y_pred (predicted labels) or score (predicted scores) or other data as inputs. Check documentation of each metric for more information.

Examples

```
>>> from secml.ml.peval.metrics import CMetric
>>> from secml.array import CArray
```

```
>>> peval = CMetric.create('accuracy')
>>> print(peval.performance_score(y_true=CArray([0, 1, 2, 3]), y_pred=CArray([0,␣
↪1, 1, 3])))
0.75
```

```
>>> peval = CMetric.create('tpr-at-fpr', fpr=0.1)
>>> print(peval.performance_score(y_true=CArray([0, 1, 0, 0]), score=CArray([1, 1,
↪ 0, 0])))
0.3
```

Attributes

best_value [best metric value. This is commonly a scalar (0.0 or 1.0).]

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.

Continued on next page

Table 98 – continued from previous page

performance_score(self[, y_true, y_pred, score])	Compute the performance metric.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

best_value = None

performance_score(*self*, *y_true=None*, *y_pred=None*, *score=None*, ***kwargs*)

Compute the performance metric.

Each metric can use as input either:

- y_true (true ground labels)
- y_pred (predicted labels)
- score (predicted scores)
- or any other data

Check documentation of each metric for more information.

If not all the required data is passed, TypeError will be raised.

CMetricAccuracy

class secml.ml.peval.metrics.c_metric_accuracy.**CMetricAccuracy**

Bases: *secml.ml.peval.metrics.c_metric.CMetric*

Performance evaluation metric: Accuracy.

Accuracy score is the percentage (inside 0/1 range) of correctly predicted labels.

The metric uses:

- y_true (true ground labels)
- y_pred (predicted labels)

Examples

```
>>> from secml.ml.peval.metrics import CMetricAccuracy
>>> from secml.array import CArray
```

```
>>> peval = CMetricAccuracy()
>>> print(peval.performance_score(CArray([0, 1, 2, 3]), CArray([0, 1, 1, 3])))
0.75
```

Attributes

class_type ['accuracy'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
performance_score(self[, y_true, y_pred, score])	Compute the performance metric.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

best_value = 1.0

CMetricAUC

class secml.ml.peval.metrics.c_metric_auc.**CMetricAUC**

Bases: *secml.ml.peval.metrics.c_metric.CMetric*

Performance evaluation metric: Area Under (ROC) Curve.

AUC is computed using the trapezoidal rule.

The metric uses:

- y_true (true ground labels)
- score (estimated target values)

Notes

This implementation is restricted to the binary classification task.

Examples

```
>>> from secml.ml.peval.metrics import CMetricAUC
>>> from secml.array import CArray
```

```
>>> peval = CMetricAUC()
>>> print(peval.performance_score(CArray([0, 1, 0, 0]), score=CArray([0, 0, 0,␣
↪0])))
0.5
```

Attributes

class_type ['auc'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
performance_score(self[, y_true, y_pred, score])	Compute the performance metric.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

best_value = 1.0

CMetricAUCWMW

class secml.ml.peval.metrics.c_metric_auc_wmw.**CMetricAUCWMW**

Bases: *secml.ml.peval.metrics.c_metric.CMetric*

Performance evaluation metric: Area Under (ROC) Curve with Wilcoxon-Mann-Whitney statistic.

The metric uses:

- y_true (true ground labels)
- score (estimated target values)

Notes

This implementation is restricted to the binary classification task.

Examples

```
>>> from secml.ml.peval.metrics import CMetricAUCWMW
>>> from secml.array import CArray
```

```
>>> peval = CMetricAUCWMW()
>>> print(peval.performance_score(CArray([0, 1, 0, 0]), score=CArray([0, 0, 0,␣
↪0])))
0.5
```

Attributes

class_type ['auc-wmw'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
performance_score(self[, y_true, y_pred, score])	Compute the performance metric.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

best_value = 1.0

CMetricConfusionMatrix

class secml.ml.peval.metrics.c_confusion_matrix.**CMetricConfusionMatrix**

Bases: *secml.ml.peval.metrics.c_metric.CMetric*

Attributes

best_value

class_type Defines class type.

logger Logger for current object.

verbose Verbosity level of logger output.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
performance_score(self[, y_true, y_pred, score])	Compute the performance metric.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

CMetricF1

class secml.ml.peval.metrics.c_metric_f1.**CMetricF1**

Bases: *secml.ml.peval.metrics.c_metric.CMetric*

Performance evaluation metric: F1.

The F1 score can be interpreted as a weighted average of the precision and recall, where an F1 score reaches its best value at 1 and worst score at 0.

The relative contribution of precision and recall to the F1 score are equal. The formula for the F1 score is:

```
F1 = 2 * (precision * recall) / (precision + recall)
```

The metric uses:

- y_true (true ground labels)
- y_pred (predicted labels)

Examples

```
>>> from secml.ml.peval.metrics import CMetricF1
>>> from secml.array import CArray
```

```
>>> peval = CMetricF1()
>>> print(peval.performance_score(CArray([0, 1, 2, 3]), CArray([0, 1, 1, 3])))
0.666666666666666
```

Attributes

class_type ['f1'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
performance_score(self[, y_true, y_pred, score])	Compute the performance metric.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

best_value = 1.0

CMetricMAE

class secml.ml.peval.metrics.c_metric_mae.**CMetricMAE**

Bases: *secml.ml.peval.metrics.c_metric.CMetric*

Performance evaluation metric: Mean Absolute Error.

Regression loss of ground truth (correct labels) and the predicted regression score.

The metric uses:

- y_true (true ground labels)
- score (estimated target values)

Examples

```
>>> from secml.ml.peval.metrics import CMetricMAE
>>> from secml.array import CArray
```

```
>>> peval = CMetricMAE()
>>> print(peval.performance_score(CArray([0, 1, 0, 0]), score=CArray([0, 0, 0,␣
↪0])))
0.25
```

Attributes

class_type ['mae'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
performance_score(self[, y_true, y_pred, score])	Compute the performance metric.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

best_value = 0.0

CMetricMSE

class secml.ml.peval.metrics.c_metric_mse.**CMetricMSE**

Bases: *secml.ml.peval.metrics.c_metric.CMetric*

Performance evaluation metric: Mean Squared Error.

Regression loss of ground truth (correct labels) and the predicted regression score.

The metric uses:

- y_true (true ground labels)
- score (estimated target values)

Examples

```
>>> from secml.ml.peval.metrics import CMetricMSE
>>> from secml.array import CArray
```

```
>>> peval = CMetricMSE()
>>> print(peval.performance_score(CArray([0, 1, 0, 0]), score=CArray([0, 0, 0,␣
↪0])))
0.25
```

Attributes

class_type ['mse'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
performance_score(self[, y_true, y_pred, score])	Compute the performance metric.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.

Continued on next page

Table 105 – continued from previous page

set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

best_value = 0.0

CMetricPartialAUC

class secml.ml.peval.metrics.c_metric_pauc.**CMetricPartialAUC**(*fpr=0.01*, *n_points=1000*)

Bases: *secml.ml.peval.metrics.c_metric.CMetric*

Performance evaluation metric: Partial Area Under (ROC) Curve.

ROC is only considered between 0 and *fpr* False Positive Rate.

AUC is computed using the trapezoidal rule.

The metric uses:

- y_true (true ground labels)
- score (estimated target values)

Notes

This implementation is restricted to the binary classification task.

Examples

```
>>> from secml.ml.peval.metrics import CMetricPartialAUC
>>> from secml.array import CArray
```

```
>>> peval = CMetricPartialAUC(fpr=0.5)
>>> print(peval.performance_score(CArray([0, 1, 0, 0]), score=CArray([0, 0, 0,␣
↪0])))
0.125
```

Attributes

class_type ['pauc'] Defines class type.

fpr [float] Desired False Positive Rate in the interval [0,1]. Default 0.01 (1%)

n_points [int] Number of points to be used when interpolating the partial ROC. Higher points means more accurate values but slower computation. Default 1000.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
performance_score(self[, y_true, y_pred, score])	Compute the performance metric.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

best_value = 1.0

CMetricPrecision

class secml.ml.peval.metrics.c_metric_precision.**CMetricPrecision**

Bases: *secml.ml.peval.metrics.c_metric.CMetric*

Performance evaluation metric: Precision.

The precision is the ratio tp / (tp + fp) where tp is the number of true positives and fp the number of false positives. The precision is intuitively the ability of the classifier not to label as positive a sample that is negative.

The metric uses:

- y_true (true ground labels)
- y_pred (predicted labels)

Examples

```
>>> from secml.ml.peval.metrics import CMetricPrecision
>>> from secml.array import CArray
```

```
>>> peval = CMetricPrecision()
>>> print(peval.performance_score(CArray([0, 1, 2, 3]), CArray([0, 1, 1, 3])))
0.625
```

Attributes

class_type ['precision'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
performance_score(self[, y_true, y_pred, score])	Compute the performance metric.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

best_value = 1.0

CMetricRecall

class secml.ml.peval.metrics.c_metric_recall.**CMetricRecall**

Bases: *secml.ml.peval.metrics.c_metric.CMetric*

Performance evaluation metric: Recall (True Positive Rate).

The recall is the ratio tp / (tp + fn) where tp is the number of true positives and fn the number of false negatives. The recall is intuitively the ability of the classifier to find all the positive samples. This is equivalent to True Positive Rate.

The metric uses:

- y_true (true ground labels)
- y_pred (predicted labels)

Examples

```
>>> from secml.ml.peval.metrics import CMetricRecall
>>> from secml.array import CArray
```

```
>>> peval = CMetricRecall()
>>> print(peval.performance_score(CArray([0, 1, 2, 3]), CArray([0, 1, 1, 3])))
0.75
```

Attributes

class_type ['recall'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
performance_score(self[, y_true, y_pred, score])	Compute the performance metric.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

best_value = 1.0

CRoc

class secml.ml.peval.metrics.c_roc.**CBaseRoc**

Bases: object

Computes the receiver operating characteristic curve, or ROC curve.

This base class manage a single classifier output (a single repetition).

See also:

CRoc class that fully supports ROC repetitions.

Attributes

fpr False Positive Rates.

th Thresholds.

tpr True Positive Rates.

Methods

compute(self, y_true, score[, positive_label])	Compute TPR/FPR for classifier output.
reset(self)	Reset stored data.

compute (*self*, *y_true*, *score*, *positive_label=None*)
Compute TPR/FPR for classifier output.

Parameters

y_true [CArray] Flat array with true binary labels in range {0, 1} for each patterns or a single array. If labels are not binary, pos_label should be explicitly given.

score [CArray] Flat array with target scores for each pattern, can either be probability estimates of the positive class or confidence values.

positive_label [int, optional] Label to consider as positive (others are considered negative).

Returns

single_roc [CBaseRoc] Instance of the roc curve (tpr, fpr, th).

property fpr
False Positive Rates.

Flat array with increasing False Positive Rates. Element i is the False Positive Rate of predictions with score >= thresholds[i].

reset (*self*)
Reset stored data.

property th
Thresholds.

Flat array with decreasing thresholds on the decision function used to compute fpr and tpr. *thresholds[0]* represents no instances being predicted and is arbitrarily set to *max(score) + 1e-3*.

property tpr
True Positive Rates.

Flat array with increasing True Positive Rates. Element i is the True Positive Rate of predictions with score >= thresholds[i].

class secml.ml.peval.metrics.c_roc.**CRoc**
Bases: *secml.ml.peval.metrics.c_roc.CBaseRoc*

Computes the receiver operating characteristic curve, or ROC curve.

"A receiver operating characteristic (ROC), or simply ROC curve, is a graphical plot which illustrates the performance of a binary classifier system as its discrimination threshold is varied. It is created by plotting the fraction of True Positive Rates out of the Positives (TPR = True Positive Rate) vs. the fraction of False Positives out of the Negatives (FPR = False Positive Rate), at various threshold settings. TPR is also known as sensitivity, and FPR is one minus the specificity or true negative rate."

The class manage different repetitions of the same classification output.

Attributes

fpr False Positive Rates.

has_mean True if average has been computed for all ROCs.

has_std_dev True if standard deviation has been computed for all ROCs.

mean_fpr Averaged False Positive Rates.

mean_tpr Averaged True Positive Rates.

n_reps Return the number of computed ROC.

std_dev_tpr Standard deviation of True Positive Rates.

th Thresholds.

tpr True Positive Rates.

Methods

average(self[, n_points, return_std])	Compute the average of computed ROC curves.
compute(self, y_true, score[, positive_label])	Compute ROC curve using input True labels and Classification Scores.
reset(self)	Reset stored data.

average (*self*, *n_points=1000*, *return_std=False*)

Compute the average of computed ROC curves.

The average ROC is reset each time *.compute_roc* is called.

Parameters

n_points [int, optional] Default 1000, is the number of points to be used for interpolation.

return_std [bool, optional] If True, standard deviation of True Positive Rates will be returned.

Returns

mean_fpr [CArray]

Flat array with increasing False Positive Rates averaged over all available repetitions. Element i is the false positive rate of predictions with score >= thresholds[i].

mean_tpr [CArray]

Flat array with increasing True Positive Rates averaged over all available repetitions. Element i is the true positive rate of predictions with score >= thresholds[i].

std_dev_tpr [CArray] Flat array with standard deviation of True Positive Rates. Only if return_std is True.

compute (*self*, *y_true*, *score*, *positive_label=None*)

Compute ROC curve using input True labels and Classification Scores.

For multi-class data, label to be considered positive should specified.

If *y_true* and *score* are both lists (with same length), one roc curve for each pair is returned. If *y_true* is a single array, one roc curve for each (y_true, score[i]) is returned.

Each time the function is called, result is appended to *tpr*,`fpr`, and *thr* class attributes. Returned ROCs are the only associated with LATEST input data.

Parameters

y_true [CArray, list] List of flat arrays with true binary labels in range {0, 1} for each patterns or a single array. If a single array, one curve is returned for each (y_true, score[i]) pair. If labels are not binary, pos_label should be explicitly given.

score [CArray, list] List of flat array with target scores for each pattern, can either be probability estimates of the positive class or confidence values. If *y_true* is a single array, one curve is returned for each (y_true, score[i]) pair.

positive_label [int, optional] Label to consider as positive (others are considered negative).

Returns

fpr [CArray or list]

Flat array with increasing False Positive Rates or a list with one array for each repetition. Element i is the False Positive Rate of predictions with score >= thresholds[i]

tpr [CArray or list]

Flat array with increasing True Positive Rates or a list with one array for each repetition. Element i is the True Positive Rate of predictions with score >= thresholds[i].

th [CArray or list]

Flat array with decreasing thresholds on the decision function used to compute fpr and tpr or a list with one array for each repetition. *thresholds[0]* represents no instances being predicted and is arbitrarily set to *max(score) + 1e-3*.

property fpr

False Positive Rates.

Flat array with increasing False Positive Rates or a list with one array for each repetition. Element i is the False Positive Rate of predictions with score >= thresholds[i].

property has_mean

True if average has been computed for all ROCs.

property has_std_dev

True if standard deviation has been computed for all ROCs.

property mean_fpr

Averaged False Positive Rates.

Flat array with increasing False Positive Rates averaged over all available repetitions. Element i is the false positive rate of predictions with score >= thresholds[i].

property mean_tpr

Averaged True Positive Rates.

Flat array with increasing True Positive Rates averaged over all available repetitions. Element i is the True Positive Rate of predictions with score >= thresholds[i].

property n_reps

Return the number of computed ROC.

property std_dev_tpr

Standard deviation of True Positive Rates.

property th

Thresholds.

Flat array with decreasing thresholds on the decision function used to compute fpr and tpr or a list with one array for each repetition. *thresholds[0]* represents no instances being predicted and is arbitrarily set to *max(score) + 1e-3*.

property tpr

True Positive Rates.

Flat array with increasing True Positive Rates or a list with one array for each repetition. Element i is the True Positive Rate of predictions with score >= thresholds[i].

secml.ml.peval.metrics.c_roc.**average**(*fpr*, *tpr*, *n_points=1000*)

Compute the average of the input tpr/fpr pairs.

Parameters

fpr, tpr [CArray or list of CArray] CArray or list of CArrays with False/True Positive Rates as output of *.CRoc*.

n_points [int, optional] Default 1000, is the number of points to be used for interpolation.

Returns

mean_fpr [CArray] Flat array with increasing False Positive Rates averaged over all available repetitions. Element i is the False Positive Rate of predictions with score >= thresholds[i].

mean_tpr [CArray] Flat array with increasing True Positive Rates averaged over all available repetitions. Element i is the True Positive Rate of predictions with score >= thresholds[i].

std_dev_tpr [CArray] Flat array with standard deviation of True Positive Rates.

secml.ml.peval.metrics.c_roc.**refine_roc**(*fpr*, *tpr*, *th*)

Function to ensure the bounds of a ROC.

The first and last points should be (0,0) and (1,1) respectively.

Parameters

fpr [CArray] False Positive Rates, as returned by *.BaseRoc.compute()*.

tpr [CArray] True Positive Rates, as returned by *.BaseRoc.compute()*.

th [CArray] Thresholds, as returned by *.BaseRoc.compute()*.

CMetricTestError

class secml.ml.peval.metrics.c_metric_test_error.**CMetricTestError**

Bases: *secml.ml.peval.metrics.c_metric.CMetric*

Performance evaluation metric: Test Error.

Test Error score is the percentage (inside 0/1 range) of wrongly predicted labels (inverse of accuracy).

The metric uses:

- y_true (true ground labels)
- y_pred (predicted labels)

Examples

```
>>> from secml.ml.peval.metrics import CMetricTestError
>>> from secml.array import CArray
```

```
>>> peval = CMetricTestError()
>>> print(peval.performance_score(CArray([0, 1, 2, 3]), CArray([0, 1, 1, 3])))
0.25
```

Attributes

class_type ['test-error'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
performance_score(self[, y_true, y_pred, score])	Compute the performance metric.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

best_value = 0.0

CMetricTPRatFPR

class secml.ml.peval.metrics.c_metric_tpr_at_fpr.**CMetricTPRatFPR**(*fpr=0.01*)

Bases: *secml.ml.peval.metrics.c_metric.CMetric*

Performance evaluation metric: True Positive Rate @ False Positive Rate.

The metric uses:

- y_true (true ground labels)
- score (estimated target values)

Notes

This implementation is restricted to the binary classification task.

Examples

```
>>> from secml.ml.peval.metrics import CMetricTPRatFPR
>>> from secml.array import CArray
```

```
>>> peval = CMetricTPRatFPR(fpr=0.5)
>>> print(peval.performance_score(CArray([0, 1, 0, 0]), score=CArray([0, 0, 0,␣
↪0])))
0.5
```

Attributes

class_type ['tpr-at-fpr'] Defines class type.

fpr [float] Desired False Positive Rate in the interval [0,1]. Default 0.01 (1%)

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
performance_score(self[, y_true, y_pred, score])	Compute the performance metric.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

best_value = 1.0

CPerfEvaluator

class secml.ml.peval.c_perfevaluator.**CPerfEvaluator**(*splitter*, *metric*)

Bases: *secml.core.c_creator.CCreator*

Evaluate the best parameters for input estimator.

Parameters

splitter [CDataSplitter or str] Object to use for splitting the dataset into train and validation.

metric [CMetric or str] Name of the metric that we want maximize / minimize.

Attributes

class_type Defines class type.

logger Logger for current object.

verbose Verbosity level of logger output.

Methods

compute_performance(self, estimator, dataset)	Compute estimator performance on input dataset.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
evaluate_params(self, estimator, dataset, ...)	Evaluate parameters for input estimator on input dataset.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

abstract compute_performance(*self*, *estimator*, *dataset*)

Compute estimator performance on input dataset.

This must be reimplemented by subclasses.

Parameters

estimator [CClassifier] The classifier that we want evaluate.

dataset [CDataset] Dataset that we want use for evaluate the classifier.

Returns

score [float] Performance score of estimator.

evaluate_params(*self*, *estimator*, *dataset*, *parameters*, *pick='first'*, *n_jobs=1*)

Evaluate parameters for input estimator on input dataset.

Parameters

estimator [CClassifier] The classifier for witch we want chose best parameters.

dataset [CDataset] Dataset to be used for evaluating parameters.

parameters [dict] Dictionary with each entry as {parameter: list of values to test}.

pick [{'first', 'last', 'random'}, optional] Defines which of the best parameters set pick. Usually, 'first' (default) correspond to the smallest parameters while 'last' correspond to the biggest. The order is consistent to the parameters dict passed as input.

n_jobs [int, optional] Number of parallel workers to use. Default 1. Cannot be higher than processor's number of cores.

Returns

best_param_dict [dict] A dictionary with the best value for each evaluated parameter.

best_value [any] Metric value obtained on validation set by the estimator.

CPerfEvaluatorXVal

class secml.ml.peval.c_perfevaluator_xval.**CPerfEvaluatorXVal**(*splitter*, *metric*)

Bases: *secml.ml.peval.c_perfevaluator.CPerfEvaluator*

Evaluate the best estimator parameters using Cross-Validation.

Parameters

splitter [CXVal or str] XVal object to be used for splitting the dataset into train and validation.

metric [CMetric or str] Name of the metric that we want maximize / minimize.

Attributes

class_type ['xval'] Defines class type.

Methods

compute_performance(self, estimator, dataset)	Split data in folds and return the mean estimator performance.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
evaluate_params(self, estimator, dataset, ...)	Evaluate parameters for input estimator on input dataset.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.

Continued on next page

Table 114 – continued from previous page

list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

compute_performance(*self*, *estimator*, *dataset*)

Split data in folds and return the mean estimator performance.

Parameters

estimator [CClassifier] The Classifier that we want evaluate

dataset [CDataset] Dataset that we want use for evaluate the classifier

Returns

score [float] Mean performance score of estimator computed on the K-Folds.

CPerfEvaluatorXValMulticlass

class secml.ml.peval.c_perfevaluator_xval_multiclass.**CPerfEvaluatorXValMulticlass**(*splitter*, *metric*)

Bases: *secml.ml.peval.c_perfevaluator.CPerfEvaluator*

Evaluate the best parameters for each single binary classifier using Cross-Validation.

Parameters

splitter [CXVal or str] XVal object to be used for splitting the dataset into train and validation.

metric [CMetric or str] Name of the metric that we want maximize / minimize.

Attributes

class_type ['xval-multiclass'] Defines class type.

Methods

compute_performance(self, estimator, dataset)	Split data in folds and return the mean estimator performance.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
evaluate_params(self, estimator, dataset, ...)	Evaluate parameters for input estimator on input dataset.
get_class_from_type(class_type)	Return the class associated with input type.

Continued on next page

Table 115 – continued from previous page

get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

compute_performance(*self*, *estimator*, *dataset*)

Split data in folds and return the mean estimator performance.

Parameters

estimator [CClassifier] The Classifier that we want evaluate

dataset [CDataset] Dataset that we want use for evaluate the classifier

Returns

scores [list] Mean performance score of each binary estimator computed on the K-Folds.

secml.ml.stats

CDensityEstimation

class secml.ml.stats.c_density_estimation.**CDensityEstimation**(*bandwidth=1.0*, *algorithm='auto'*, *kernel='gaussian'*, *metric='euclidean'*, *atol=0*, *rtol=1e-08*, *breadth_first=True*, *leaf_size=40*, *metric_params=None*)

Bases: *secml.core.c_creator.CCreator*

Kernel Density Estimation

Parameters

bandwidth [float, optional] The bandwidth of the kernel. Default 1.

algorithm [str, optional] The tree algorithm to use. Valid options are ['kd_tree'|'ball_tree'|'auto']. Default is 'auto'.

kernel [str, optional] The kernel to use. Valid kernels are ['gaussian'|'tophat'|'epanechnikov'|'exponential'|'linear'|'cosine']. Default is 'gaussian'.

metric [str, optional] The distance metric to use. Note that not all metrics are valid with all algorithms. Refer to the documentation of BallTree and KDTree for a description of avail-

able algorithms. Note that the normalization of the density output is correct only for the Euclidean distance metric. Default is 'euclidean'.

atol [float, optional] The desired absolute tolerance of the result. A larger tolerance will generally lead to faster execution. Default is 0.

rtol [float, optional] The desired relative tolerance of the result. A larger tolerance will generally lead to faster execution. Default is 1E-8.

breadth_first [bool, optional] If true (default), use a breadth-first approach to the problem. Otherwise use a depth-first approach.

leaf_size [int, optional] Specify the leaf size of the underlying tree. See BallTree or KDTree for details. Default is 40.

metric_params [dict, optional] Additional parameters to be passed to the tree for use with the metric. For more information, see the documentation of BallTree or KDTree.

Attributes

class_type Defines class type.

logger Logger for current object.

verbose Verbosity level of logger output.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
estimate_density(self, x[, n_points])	Estimate density of input array.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

estimate_density(*self*, *x*, *n_points=1000*)

Estimate density of input array.

Returns

x [CArray] Arrays with coordinates used to estimate density.

df [CArray] Density function values.

CDistributionGaussian

class secml.ml.stats.c_distribution_gaussian.**CDistributionGaussian**(*mean=0*, *cov=1*)

Bases: *secml.core.c_creator.CCreator*

A multivariate normal random variable.

Parameters

mean [scalar, optional] Mean of the distribution (default zero)

cov [array_like or scalar, optional] Covariance matrix of the distribution (default one)

Attributes

class_type Defines class type.

logger Logger for current object.

verbose Verbosity level of logger output.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
logpdf(self, data)	Log of the probability density function.
pdf(self, data)	Probability density function.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

logpdf(*self*, *data*)

Log of the probability density function.

Parameters

data [CArray] Quantiles, with the last axis of x denoting the components.

Returns

pdf: CArray Probability density function computed at input data.

pdf(*self*, *data*)

Probability density function.

Parameters

data [CArray] Quantiles, with the last axis of x denoting the components.

Returns

pdf: CArray Probability density function computed at input data.

4.7.8 secml.adv

Adversarial Machine Learning

secml.adv.attacks

secml.adv.attacks.evasion

CAttackEvasion

class secml.adv.attacks.evasion.c_attack_evasion.**CAttackEvasion**(*classifier*, *y_target=None*, *attack_classes='all'*)

Bases: *secml.adv.attacks.c_attack.CAttack*

Interface class for evasion and poisoning attacks.

Parameters

classifier [CClassifier] Target classifier (trained).

y_target [int or None, optional] If None an error-generic attack will be performed, else a error-specific attack to have the samples misclassified as belonging to the *y_target* class.

attack_classes ['all' or CArray, optional]

Array with the classes that can be manipulated by the attacker or 'all' (default) if all classes can be manipulated.

Attributes

attack_classes

class_type Defines class type.

classifier Returns classifier

f_eval Returns the number of function evaluations made during the attack.

f_opt Returns the value of the objective function evaluated on the optimal point founded by the attack.

f_seq Returns a CArray containing the values of the objective function evaluations made by the attack.

grad_eval Returns the number of gradient evaluations made during the attack.

logger Logger for current object.

verbose Verbosity level of logger output.

x_opt Returns the optimal point founded by the attack.

x_seq Returns a CArray (number of iteration * number of features) containing the values of the attack point path.

y_target

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
is_attack_class(self, y)	Returns True/False if the input class can be attacked.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
objective_function(self, x)	Objective function.
objective_function_gradient(self, x)	Gradient of the objective function.
run(self, x, y[, ds_init])	Runs evasion on a dataset.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property attack_classes

is_attack_class(*self*, *y*)

Returns True/False if the input class can be attacked.

Parameters

y [int or CArray] CArray or single label of the class to to be checked.

Returns

bool or CArray

True if class y can be manipulated by the attacker, False otherwise. If CArray, a True/False value for each input label will be returned.

abstract objective_function(*self*, *x*)

Objective function.

Parameters

x [CArray or CDataset]

Returns

f_obj [float or CArray of floats]

abstract objective_function_gradient(*self*, *x*)
Gradient of the objective function.

run(*self*, *x*, *y*, *ds_init=None*)
Runs evasion on a dataset.

Parameters

x [CArray] Data points.

y [CArray] True labels.

ds_init [CDataset] Dataset for warm starts.

Returns

y_pred [CArray] Predicted labels for all ds samples by target classifier.

scores [CArray] Scores for all ds samples by target classifier.

adv_ds [CDataset] Dataset of manipulated samples.

f_obj [float] Mean value of the objective function computed on each data point.

property y_target

CAttackEvasionPGD

class secml.adv.attacks.evasion.c_attack_evasion_pgd.**CAttackEvasionPGD**(*classifier*, *double_init_ds=None*, *double_init=True*, *distance='l1'*, *dmax=0*, *lb=0*, *ub=1*, *discrete=<no value>*, *y_target=None*, *attack_classes='all'*, *solver_params=None*)

Bases: *secml.adv.attacks.evasion.c_attack_evasion_pgd_ls.CAttackEvasionPGDLS*

Evasion attacks using Projected Gradient Descent.

This class implements the maximum-confidence evasion attacks proposed in:

- https://arxiv.org/abs/1708.06939, ICCV W. ViPAR, 2017.

This is the multi-class extension of our original work in:

- https://arxiv.org/abs/1708.06131, ECML 2013, implemented using a standard projected gradient solver.

It can also be used on sparse, high-dimensional feature spaces, using an L1 constraint on the manipulation of samples to preserve sparsity, as we did for crafting adversarial Android malware in:

- https://arxiv.org/abs/1704.08996, IEEE TDSC 2017.

For more on evasion attacks, see also:

- https://arxiv.org/abs/1809.02861, USENIX Sec. 2019
- https://arxiv.org/abs/1712.03141, Patt. Rec. 2018

Parameters

classifier [CClassifier] Target classifier.

double_init_ds [CDataset or None, optional] Dataset used to initialize an alternative init point (double init).

double_init [bool, optional] If True (default), use double initialization point. Needs double_init_ds not to be None.

distance [{'l1' or 'l2'}, optional] Norm to use for computing the distance of the adversarial example from the original sample. Default 'l2'.

dmax [scalar, optional] Maximum value of the perturbation. Default 1.

lb, ub [int or CArray, optional] Lower/Upper bounds. If int, the same bound will be applied to all the features. If CArray, a different bound can be specified for each feature. Default *lb = 0*, *ub = 1*.

y_target [int or None, optional] If None an error-generic attack will be performed, else a error-specific attack to have the samples misclassified as belonging to the *y_target* class.

attack_classes ['all' or CArray, optional]

Array with the classes that can be manipulated by the attacker or 'all' (default) if all classes can be manipulated.

solver_params [dict or None, optional] Parameters for the solver. Default None, meaning that default parameters will be used.

Attributes

`class_type` ['e-pgd'] Defines class type.

Methods

`copy`(self)	Returns a shallow copy of current class.
`create`([class_item])	This method creates an instance of a class with given type.
`deepcopy`(self)	Returns a deep copy of current class.
`get_class_from_type`(class_type)	Return the class associated with input type.
`get_params`(self)	Returns the dictionary of class hyperparameters.
`get_state`(self)	Returns the object state dictionary.
`get_subclasses`()	Get all the subclasses of the calling class.
`is_attack_class`(self, y)	Returns True/False if the input class can be attacked.
`list_class_types`()	This method lists all types of available subclasses of calling one.
`load`(path)	Loads object from file.
`load_state`(self, path)	Sets the object state from file.
`objective_function`(self, x)	Compute the objective function of the evasion attack.
`objective_function_gradient`(self, x)	Compute the gradient of the evasion objective function.

Continued on next page

Table 119 – continued from previous page

run(self, x, y[, ds_init])	Runs evasion on a dataset.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

CAttackEvasionPGDLS

class secml.adv.attacks.evasion.c_attack_evasion_pgd_ls.**CAttackEvasionPGDLS**(*classifier*, *double_init_ds=None*, *double_init=True*, *distance='l1'*, *dmax=0*, *lb=0*, *ub=1*, *y_target=None*, *attack_classes='all'*, *solver_params=None*

Bases: *secml.adv.attacks.evasion.c_attack_evasion.CAttackEvasion*, secml.adv.attacks.c_attack_mixin.CAttackMixin

Evasion attacks using Projected Gradient Descent with Line Search.

This class implements the maximum-confidence evasion attacks proposed in:

- https://arxiv.org/abs/1708.06939, ICCV W. ViPAR, 2017.

This is the multi-class extension of our original work in:

- https://arxiv.org/abs/1708.06131, ECML 2013,

implemented using a custom projected gradient solver that uses line search in each iteration to save gradient computations and speed up the attack.

It can also be used on sparse, high-dimensional feature spaces, using an L1 constraint on the manipulation of samples to preserve sparsity, as we did for crafting adversarial Android malware in:

- https://arxiv.org/abs/1704.08996, IEEE TDSC 2017.

For more on evasion attacks, see also:

- https://arxiv.org/abs/1809.02861, USENIX Sec. 2019
- https://arxiv.org/abs/1712.03141, Patt. Rec. 2018

Parameters

classifier [CClassifier] Target classifier.

double_init_ds [CDataset or None, optional] Dataset used to initialize an alternative init point (double init).

double_init [bool, optional] If True (default), use double initialization point. Needs double_init_ds not to be None.

distance [{'l1' or 'l2'}, optional] Norm to use for computing the distance of the adversarial example from the original sample. Default 'l2'.

dmax [scalar, optional] Maximum value of the perturbation. Default 1.

lb, ub [int or CArray, optional] Lower/Upper bounds. If int, the same bound will be applied to all the features. If CArray, a different bound can be specified for each feature. Default *lb = 0*, *ub = 1*.

y_target [int or None, optional] If None an error-generic attack will be performed, else a error-specific attack to have the samples misclassified as belonging to the *y_target* class.

attack_classes ['all' or CArray, optional]

Array with the classes that can be manipulated by the attacker or 'all' (default) if all classes can be manipulated.

solver_params [dict or None, optional] Parameters for the solver. Default None, meaning that default parameters will be used.

Attributes

class_type ['e-pgd-ls'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
is_attack_class(self, y)	Returns True/False if the input class can be attacked.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
objective_function(self, x)	Compute the objective function of the evasion attack.
objective_function_gradient(self, x)	Compute the gradient of the evasion objective function.
run(self, x, y[, ds_init])	Runs evasion on a dataset.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property double_init

property double_init_ds

Returns the CDataset used for the double initialization

objective_function (*self*, *x*)
Compute the objective function of the evasion attack.

The objective function is:

- **for error-generic attack:** min f_obj(x) = f_{klo (if the sample is rejected) }(x) argmax_{(c != k) and (c != o)} f_c(x), where k is the true class, o is the reject class and c is the competing class, which is the class with the maximum score, and can be neither k nor c

-for error-specific attack: min -f_obj(x) = -f_k(x) + argmax_{c != k} f_c(x), where k is the target class and c is the competing class, which is the class with the maximum score except for the target class

Parameters

x [CArray] Array containing the data points (one or more than one).

Returns

f_obj [CArray] Values of objective function at x.

objective_function_gradient (*self*, *x*)
Compute the gradient of the evasion objective function.

Parameters

x [CArray] A single point.

property y_target

CAttackEvasionPGDExp

class secml.adv.attacks.evasion.c_attack_evasion_pgd_exp.**CAttackEvasionPGDExp**(*classifier*, *double_init_ds=None*, *double_init=True*, *distance='l1'*, *dmax=0*, *lb=0*, *ub=1*, *y_target=None*, *attack_classes='all'*, *solver_params=N*

Bases: *secml.adv.attacks.evasion.c_attack_evasion_pgd_ls.CAttackEvasionPGDLS*

Evasion attacks using Projected Gradient Descent with Exponential line search.

This class implements the maximum-confidence evasion attacks proposed in:

- https://arxiv.org/abs/1910.00470, EURASIP JIS, 2020.
- https://arxiv.org/abs/1708.06939, ICCV W. ViPAR, 2017.

It is the multi-class extension of our original work in:

- https://arxiv.org/abs/1708.06131, ECML 2013, implemented using a standard projected gradient solver.

This attack uses a faster line search than PGD-LS.

In all our attacks, we use a smart double initialization to avoid using the mimicry term from our ECML 2013 paper, as described in: - https://pralab.diee.unica.it/sites/default/files/zhang15-tcyb.pdf, IEEE TCYB, 2015

If the attack is not successful when starting from x0, we initialize the optimization by projecting a point from another class onto the feasible domain and try again.

Parameters

classifier [CClassifier] Target classifier.

double_init_ds [CDataset or None, optional] Dataset used to initialize an alternative init point (double init).

double_init [bool, optional] If True (default), use double initialization point. Needs double_init_ds not to be None.

distance [{'l1' or 'l2'}, optional] Norm to use for computing the distance of the adversarial example from the original sample. Default 'l2'.

dmax [scalar, optional] Maximum value of the perturbation. Default 1.

lb, ub [int or CArray, optional] Lower/Upper bounds. If int, the same bound will be applied to all the features. If CArray, a different bound can be specified for each feature. Default *lb = 0*, *ub = 1*.

y_target [int or None, optional] If None an error-generic attack will be performed, else a error-specific attack to have the samples misclassified as belonging to the *y_target* class.

attack_classes ['all' or CArray, optional]

Array with the classes that can be manipulated by the attacker or 'all' (default) if all classes can be manipulated.

solver_params [dict or None, optional] Parameters for the solver. Default None, meaning that default parameters will be used.

Attributes

`class_type` ['e-pgd-exp'] Defines class type.

Methods

`copy`(self)	Returns a shallow copy of current class.
`create`([class_item])	This method creates an instance of a class with given type.
`deepcopy`(self)	Returns a deep copy of current class.
`get_class_from_type`(class_type)	Return the class associated with input type.
`get_params`(self)	Returns the dictionary of class hyperparameters.
`get_state`(self)	Returns the object state dictionary.
`get_subclasses`()	Get all the subclasses of the calling class.
`is_attack_class`(self, y)	Returns True/False if the input class can be attacked.
`list_class_types`()	This method lists all types of available subclasses of calling one.
`load`(path)	Loads object from file.
`load_state`(self, path)	Sets the object state from file.
`objective_function`(self, x)	Compute the objective function of the evasion attack.

Continued on next page

Table 121 – continued from previous page

objective_function_gradient(self, x)	Compute the gradient of the evasion objective function.
run(self, x, y[, ds_init])	Runs evasion on a dataset.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

CAttackEvasionCleverhans

class secml.adv.attacks.evasion.cleverhans.c_attack_evasion_cleverhans.**CAttackEvasionCleve**

Bases: *secml.adv.attacks.evasion.c_attack_evasion.CAttackEvasion*, secml.adv.attacks.evasion.cleverhans.c_attack_evasion_cleverhans_losses.CAttackEvasionCleverhansLossesMixin

This class is a wrapper of the attacks implemented in the Cleverhans library.

Credits: https://github.com/tensorflow/cleverhans.

Parameters

classifier [CClassifier] Target classifier (trained).

y_target [int or None, optional] If None an indiscriminate attack will be performed, else a targeted attack to have the samples misclassified as belonging to the y_target class.

clvh_attack_class: The CleverHans class that implement the attack

store_var_list: list list of variables to store from the graph during attack run. The variables will be stored as key-value dictionary and can be retrieved through the property *stored_vars*.

****kwargs** Any other parameter for the cleverhans attack.

Notes

The current Tensorflow default graph will be used.

Attributes

attack_classes

attack_params Object containing all Cleverhans parameters

class_type Defines class type.

classifier Returns classifier

f_eval Returns the number of function evaluations made during the attack.

f_opt Returns the value of the objective function evaluated on the optimal point founded by the attack.

f_seq Returns a CArray containing the values of the objective function evaluations made by the attack.

grad_eval Returns the number of gradient evaluations made during the attack.

logger Logger for current object.

stored_vars Variables extracted from the graph during execution of the attack.

verbose Verbosity level of logger output.

x_opt Returns the optimal point founded by the attack.

x_seq Returns a CArray (number of iteration * number of features) containing the values of the attack point path.

y_target

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
is_attack_class(self, y)	Returns True/False if the input class can be attacked.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
objective_function(self, x)	Objective function.
objective_function_gradient(self, x)	Gradient of the objective function.
run(self, x, y[, ds_init])	Runs evasion on a dataset.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property attack_params
Object containing all Cleverhans parameters

property f_eval
Returns the number of function evaluations made during the attack.

property grad_eval
Returns the number of gradient evaluations made during the attack.

objective_function(*self*, *x*)
Objective function.

Parameters

x [CArray or CDataset]

Returns

f_obj [float or CArray of floats]

objective_function_gradient (*self*, *x*)
Gradient of the objective function.

run (*self*, *x*, *y*, *ds_init=None*)
Runs evasion on a dataset.

Parameters

x [CArray] Data points.

y [CArray] True labels.

ds_init [CDataset] Dataset for warm starts.

Returns

y_pred [CArray] Predicted labels for all ds samples by target classifier.

scores [CArray] Scores for all ds samples by target classifier.

adv_ds [CDataset] Dataset of manipulated samples.

f_obj [float] Mean value of the objective function computed on each data point.

set (*self*, *param_name*, *param_value*, *copy=False*)
Set a parameter of the class.

Only writable attributes of the class, i.e. PUBLIC or READ/WRITE, can be set.

The following checks are performed before setting:

- if *param_name* is an attribute of current class, set directly;
- **else, iterate over __dict__ and look for a class attribute** having the desired parameter as an attribute;
- **else, if attribute is not found on the 2nd level,** raise AttributeError.

If possible, a reference to the attribute to set is assigned. Use *copy=True* to always make a deepcopy before set.

Parameters

param_name [str] Name of the parameter to set.

param_value [any] Value to set for the parameter.

copy [bool] By default (False) a reference to the parameter to assign is set. If True or a reference cannot be extracted, a deepcopy of the parameter value is done first.

property stored_vars
Variables extracted from the graph during execution of the attack.

secml.adv.attacks.poisoning

CAttackPoisoning

class secml.adv.attacks.poisoning.c_attack_poisoning.**CAttackPoisoning**(*classifier*, *training_data*, *val*, *distance='l2'*, *dmax=0*, *lb=0*, *ub=1*, *y_target=None*, *solver_type='pgd-ls'*, *solver_params=None*, *init_type='random'*, *random_seed=None*)

Bases: secml.adv.attacks.c_attack_mixin.CAttackMixin

Interface for poisoning attacks.

Parameters

classifier [CClassifier] Target classifier.

training_data [CDataset] Dataset on which the the classifier has been trained on.

val [CDataset] Validation set.

distance [{'l1' or 'l2'}, optional] Norm to use for computing the distance of the adversarial example from the original sample. Default 'l2'.

dmax [scalar, optional] Maximum value of the perturbation. Default 1.

lb, ub [int or CArray, optional] Lower/Upper bounds. If int, the same bound will be applied to all the features. If CArray, a different bound can be specified for each feature. Default *lb = 0*, *ub = 1*.

y_target [int or None, optional] If None an error-generic attack will be performed, else a error-specific attack to have the samples misclassified as belonging to the *y_target* class.

solver_type [str or None, optional] Identifier of the solver to be used. Default 'pgd-ls'.

solver_params [dict or None, optional] Parameters for the solver. Default None, meaning that default parameters will be used.

init_type [{'random', 'loss_based'}, optional] Strategy used to chose the initial random samples. Default 'random'.

random_seed [int or None, optional] If int, random_state is the seed used by the random number generator. If None, no fixed seed will be set.

Attributes

class_type Defines class type.

classifier Returns classifier

distance todo

dmax Returns dmax

f_eval Returns the number of function evaluations made during the attack.

f_opt Returns the value of the objective function evaluated on the optimal point founded by the attack.

f_seq Returns a CArray containing the values of the objective function evaluations made by the attack.

grad_eval Returns the number of function evaluations made during the attack.

lb Returns lb

logger Logger for current object.

n_points Returns the number of poisoning points.

random_seed Returns the attacker's validation data

solver_params

solver_type

training_data Returns the training set used to learn the targeted classifier

ub Returns ub

val Returns the attacker's validation data

verbose Verbosity level of logger output.

x0 Returns the attacker's initial sample features

x_opt Returns the optimal point founded by the attack.

x_seq Returns a CArray (number of iteration * number of features) containing the values of the attack point path.

xc Returns the attacker's sample features

y_target

yc Returns the attacker's sample label

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.

Continued on next page

Table 123 – continued from previous page

objective_function(self, xc[, acc])	**Parameters**
objective_function_gradient(self, xc[, ...])	Compute the loss derivative wrt the attack sample xc
run(self, x, y[, ds_init, max_iter])	Runs poisoning on multiple points.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property n_points
Returns the number of poisoning points.

objective_function(*self*, *xc*, *acc=False*)

Parameters

xc: poisoning point

Returns

f_obj: values of objective function (average hinge loss) at x

objective_function_gradient(*self*, *xc*, *normalization=True*)
Compute the loss derivative wrt the attack sample xc

The derivative is decomposed as:

dl / x = sum^n_c=1 (dl / df_c * df_c / x)

property random_seed
Returns the attacker's validation data

run(*self*, *x*, *y*, *ds_init=None*, *max_iter=1*)
Runs poisoning on multiple points.

It reads n_points (previously set), initializes xc, yc at random, and then optimizes the poisoning points xc.

Parameters

x [CArray] Validation set for evaluating classifier performance. Note that this is not the validation data used by the attacker, which should be passed instead to *CAttackPoisoning* init.

y [CArray] Corresponding true labels for samples in *x*.

ds_init [CDataset or None, optional.] Dataset for warm start.

max_iter [int, optional] Number of iterations to re-optimize poisoning data. Default 1.

Returns

y_pred [predicted labels for all val samples by targeted classifier]

scores [scores for all val samples by targeted classifier]

adv_xc [manipulated poisoning points xc (for subsequents warm starts)]

f_opt [final value of the objective function]

property training_data
Returns the training set used to learn the targeted classifier

property val
Returns the attacker's validation data

property x0
Returns the attacker's initial sample features

property xc
Returns the attacker's sample features

property y_target

property yc
Returns the attacker's sample label

CAttackPoisoningLogisticRegression

class secml.adv.attacks.poisoning.c_attack_poisoning_logistic_regression.**CAttackPoisoningLo**

Bases: *secml.adv.attacks.poisoning.c_attack_poisoning.CAttackPoisoning*

Poisoning attacks against logistic regression.

This is an implementation of the attack developed in Sect. 3.3 in https://www.usenix.org/conference/usenixsecurity19/presentation/demontis:

- A. Demontis, M. Melis, M. Pintor, M. Jagielski, B. Biggio, A. Oprea, C. Nita-Rotaru, and F. Roli. Why do adversarial attacks transfer? Explaining transferability of evasion and poisoning attacks. In 28th USENIX Security Symposium. USENIX Association, 2019.

For more details on poisoning attacks, see also:

- https://arxiv.org/abs/1804.00308, IEEE Symp. SP 2018
- https://arxiv.org/abs/1712.03141, Patt. Rec. 2018
- https://arxiv.org/abs/1708.08689, AISec 2017
- https://arxiv.org/abs/1804.07933, ICML 2015
- https://arxiv.org/pdf/1206.6389, ICML 2012

Parameters

classifier [CClassifierLogistic] Target classifier.

training_data [CDataset] Dataset on which the the classifier has been trained on.

val [CDataset] Validation set.

distance [{'l1' or 'l2'}, optional] Norm to use for computing the distance of the adversarial example from the original sample. Default 'l2'.

dmax [scalar, optional] Maximum value of the perturbation. Default 1.

lb, ub [int or CArray, optional] Lower/Upper bounds. If int, the same bound will be applied to all the features. If CArray, a different bound can be specified for each feature. Default *lb = 0*, *ub = 1*.

y_target [int or None, optional] If None an error-generic attack will be performed, else a error-specific attack to have the samples misclassified as belonging to the *y_target* class.

solver_type [str or None, optional] Identifier of the solver to be used. Default 'pgd-ls'.

solver_params [dict or None, optional] Parameters for the solver. Default None, meaning that default parameters will be used.

init_type [{'random', 'loss_based'}, optional] Strategy used to chose the initial random samples. Default 'random'.

random_seed [int or None, optional] If int, random_state is the seed used by the random number generator. If None, no fixed seed will be set.

Attributes

`class_type` Defines class type.

`classifier` Returns classifier

`distance` todo

`dmax` Returns dmax

`f_eval` Returns the number of function evaluations made during the attack.

`f_opt` Returns the value of the objective function evaluated on the optimal point founded by the attack.

`f_seq` Returns a CArray containing the values of the objective function evaluations made by the attack.

`grad_eval` Returns the number of function evaluations made during the attack.

`lb` Returns lb

`logger` Logger for current object.

`n_points` Returns the number of poisoning points.

`random_seed` Returns the attacker's validation data

solver_params

solver_type

`training_data` Returns the training set used to learn the targeted classifier

`ub` Returns ub

`val` Returns the attacker's validation data

`verbose` Verbosity level of logger output.

`x0` Returns the attacker's initial sample features

x_opt Returns the optimal point founded by the attack.

x_seq Returns a CArray (number of iteration * number of features) containing the values of the attack point path.

xc Returns the attacker's sample features

y_target

yc Returns the attacker's sample label

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
objective_function(self, xc[, acc])	**Parameters**
objective_function_gradient(self, xc[, ...])	Compute the loss derivative wrt the attack sample xc
run(self, x, y[, ds_init, max_iter])	Runs poisoning on multiple points.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

CAttackPoisoningRidge

class secml.adv.attacks.poisoning.c_attack_poisoning_ridge.**CAttackPoisoningRidge**(*classifier*, *training_data*, *val*, *distance='l2'*, *dmax=0*, *lb=0*, *ub=1*, *y_target=Non*, *solver_type='*, *ls'*, *solver_param*, *init_type=Nor*, *random_seed=No*

Bases: *secml.adv.attacks.poisoning.c_attack_poisoning.CAttackPoisoning*

Poisoning attacks against ridge regression.

This is an implementation of the attack developed in https://arxiv.org/abs/1804.07933:

- H. Xiao, B. Biggio, G. Brown, G. Fumera, C. Eckert, and F. Roli. Is feature selection secure against training data poisoning? In F. Bach and D. Blei, editors, JMLR W&CP, Proc. 32nd Int'l Conf. Mach. Learning (ICML), volume 37, pp. 1689-1698, 2015.

For more details on poisoning attacks, see also:

- https://arxiv.org/abs/1809.02861, USENIX Sec. 2019
- https://arxiv.org/abs/1804.00308, IEEE Symp. SP 2018
- https://arxiv.org/abs/1712.03141, Patt. Rec. 2018
- https://arxiv.org/abs/1708.08689, AISec 2017
- https://arxiv.org/pdf/1206.6389, ICML 2012

Parameters

classifier [CClassifierRidge] Target classifier.

training_data [CDataset] Dataset on which the the classifier has been trained on.

val [CDataset] Validation set.

distance [{'l1' or 'l2'}, optional] Norm to use for computing the distance of the adversarial example from the original sample. Default 'l2'.

dmax [scalar, optional] Maximum value of the perturbation. Default 1.

lb, ub [int or CArray, optional] Lower/Upper bounds. If int, the same bound will be applied to all the features. If CArray, a different bound can be specified for each feature. Default *lb = 0*, *ub = 1*.

y_target [int or None, optional] If None an error-generic attack will be performed, else a error-specific attack to have the samples misclassified as belonging to the *y_target* class.

solver_type [str or None, optional] Identifier of the solver to be used. Default 'pgd-ls'.

solver_params [dict or None, optional] Parameters for the solver. Default None, meaning that default parameters will be used.

init_type [{'random', 'loss_based'}, optional] Strategy used to chose the initial random samples. Default 'random'.

random_seed [int or None, optional] If int, random_state is the seed used by the random number generator. If None, no fixed seed will be set.

Attributes

class_type Defines class type.

classifier Returns classifier

distance todo

dmax Returns dmax

f_eval Returns the number of function evaluations made during the attack.

f_opt Returns the value of the objective function evaluated on the optimal point founded by the attack.

f_seq Returns a CArray containing the values of the objective function evaluations made by the attack.

grad_eval Returns the number of function evaluations made during the attack.

lb Returns lb

logger Logger for current object.

n_points Returns the number of poisoning points.

random_seed Returns the attacker's validation data

solver_params

solver_type

training_data Returns the training set used to learn the targeted classifier

ub Returns ub

val Returns the attacker's validation data

verbose Verbosity level of logger output.

x0 Returns the attacker's initial sample features

x_opt Returns the optimal point founded by the attack.

x_seq Returns a CArray (number of iteration * number of features) containing the values of the attack point path.

xc Returns the attacker's sample features

y_target

yc Returns the attacker's sample label

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
objective_function(self, xc[, acc])	**Parameters**
objective_function_gradient(self, xc[, ...])	Compute the loss derivative wrt the attack sample xc
run(self, x, y[, ds_init, max_iter])	Runs poisoning on multiple points.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

CAttackPoisoningSVM

class secml.adv.attacks.poisoning.c_attack_poisoning_svm.**CAttackPoisoningSVM**(*classifier*, *training_data*, *val*, *distance='l1'*, *dmax=0*, *lb=0*, *ub=1*, *y_target=None*, *solver_type='pgd-ls'*, *solver_params=Nor*
init_type='random', *random_seed=None*)

Bases: *secml.adv.attacks.poisoning.c_attack_poisoning.CAttackPoisoning*

Poisoning attacks against Support Vector Machines (SVMs).

This is an implementation of the attack in https://arxiv.org/pdf/1206.6389:

- B. Biggio, B. Nelson, and P. Laskov. Poisoning attacks against support vector machines. In J. Langford

and J. Pineau, editors, 29th Int'l Conf. on Machine Learning, pages 1807-1814. Omnipress, 2012.

where the gradient is computed as described in Eq. (10) in https://www.usenix.org/conference/usenixsecurity19/presentation/demontis:

- A. Demontis, M. Melis, M. Pintor, M. Jagielski, B. Biggio, A. Oprea, C. Nita-Rotaru, and F. Roli. Why do adversarial attacks transfer? Explaining transferability of evasion and poisoning attacks. In 28th USENIX Security Symposium. USENIX Association, 2019.

For more details on poisoning attacks, see also:

- https://arxiv.org/abs/1804.00308, IEEE Symp. SP 2018
- https://arxiv.org/abs/1712.03141, Patt. Rec. 2018
- https://arxiv.org/abs/1708.08689, AISec 2017
- https://arxiv.org/abs/1804.07933, ICML 2015

Parameters

classifier [CClassifierSVM] Target SVM, trained in the dual (i.e., with kernel not set to None).

training_data [CDataset] Dataset on which the the classifier has been trained on.

val [CDataset] Validation set.

distance [{'l1' or 'l2'}, optional] Norm to use for computing the distance of the adversarial example from the original sample. Default 'l2'.

dmax [scalar, optional] Maximum value of the perturbation. Default 1.

lb, ub [int or CArray, optional] Lower/Upper bounds. If int, the same bound will be applied to all the features. If CArray, a different bound can be specified for each feature. Default *lb = 0*, *ub = 1*.

y_target [int or None, optional] If None an error-generic attack will be performed, else a error-specific attack to have the samples misclassified as belonging to the *y_target* class.

solver_type [str or None, optional] Identifier of the solver to be used. Default 'pgd-ls'.

solver_params [dict or None, optional] Parameters for the solver. Default None, meaning that default parameters will be used.

init_type [{'random', 'loss_based'}, optional] Strategy used to chose the initial random samples. Default 'random'.

random_seed [int or None, optional] If int, random_state is the seed used by the random number generator. If None, no fixed seed will be set.

Attributes

`class_type` Defines class type.

`classifier` Returns classifier

`distance` todo

`dmax` Returns dmax

`f_eval` Returns the number of function evaluations made during the attack.

`f_opt` Returns the value of the objective function evaluated on the optimal point founded by the attack.

`f_seq` Returns a CArray containing the values of the objective function evaluations made by the attack.

grad_eval Returns the number of function evaluations made during the attack.

lb Returns lb

logger Logger for current object.

n_points Returns the number of poisoning points.

random_seed Returns the attacker's validation data

solver_params

solver_type

training_data Returns the training set used to learn the targeted classifier

ub Returns ub

val Returns the attacker's validation data

verbose Verbosity level of logger output.

x0 Returns the attacker's initial sample features

x_opt Returns the optimal point founded by the attack.

x_seq Returns a CArray (number of iteration * number of features) containing the values of the attack point path.

xc Returns the attacker's sample features

y_target

yc Returns the attacker's sample label

Methods

alpha_xc(self, xc)	**Parameters**
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
objective_function(self, xc[, acc])	**Parameters**
objective_function_gradient(self, xc[, ...])	Compute the loss derivative wrt the attack sample xc
run(self, x, y[, ds_init, max_iter])	Runs poisoning on multiple points.
save(self, path)	Save class object to file.

Continued on next page

Table 126 – continued from previous page

save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

alpha_xc(*self*, *xc*)

Parameters

xc: poisoning point

Returns

f_obj: values of objective function (average hinge loss) at x

CAttack

class secml.adv.attacks.c_attack.**CAttack**(*classifier*)

Bases: *secml.core.c_creator.CCreator*

Generic interface class for adversarial attacks.

Parameters

classifier [CClassifier] Target classifier.

Attributes

class_type Defines class type.

classifier Returns classifier

f_eval Returns the number of function evaluations made during the attack.

f_opt Returns the value of the objective function evaluated on the optimal point founded by the attack.

f_seq Returns a CArray containing the values of the objective function evaluations made by the attack.

grad_eval Returns the number of gradient evaluations made during the attack.

logger Logger for current object.

verbose Verbosity level of logger output.

x_opt Returns the optimal point founded by the attack.

x_seq Returns a CArray (number of iteration * number of features) containing the values of the attack point path.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
run(self, x, y[, ds_init])	Run attack on the dataset x,y (with multiple attack points).
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property classifier
Returns classifier

abstract property f_eval
Returns the number of function evaluations made during the attack.

property f_opt
Returns the value of the objective function evaluated on the optimal point founded by the attack.

> **Warning:** Due to a known issue, if more then one sample is passed to `.run()`, this property will only return the data relative to the last optimized one. This behavior will change in a future version.

property f_seq
Returns a CArray containing the values of the objective function evaluations made by the attack.

> **Warning:** Due to a known issue, if more then one sample is passed to `.run()`, this property will only return the data relative to the last optimized one. This behavior will change in a future version.

abstract property grad_eval
Returns the number of gradient evaluations made during the attack.

abstract run (*self*, *x*, *y*, *ds_init=None*)
Run attack on the dataset x,y (with multiple attack points).

Parameters

x [CArray] Initial samples.

y [int or CArray] The true label of x.

ds_init [CDataset or None, optional.] Dataset for warm start.

Returns

y_pred [predicted labels for all samples by the targeted classifier]

scores [scores for all samples by targeted classifier]

adv_ds [manipulated attack samples (for subsequents warm starts)]

f_opt [final value of the objective function]

property x_opt

Returns the optimal point founded by the attack.

> **Warning:** Due to a known issue, if more then one sample is passed to `.run()`, this property will only return the data relative to the last optimized one. This behavior will change in a future version.

property x_seq

Returns a CArray (number of iteration * number of features) containing the values of the attack point path.

> **Warning:** Due to a known issue, if more then one sample is passed to `.run()`, this property will only return the data relative to the last optimized one. This behavior will change in a future version.

secml.adv.seceval

CSecEval

class secml.adv.seceval.c_sec_eval.**CSecEval**(*attack*, *param_name*, *param_values*, *save_adv_ds=False*)

Bases: *secml.core.c_creator.CCreator*

This class repeat the security evaluation (where security is measured with a given metric) while the power of the attacker increase.

Parameters

attack [CAttack] Class that implements an attack (e.g evasion or poisoning)

param_name [str] Name of the parameter that represents the increasingly attacker power.

param_values [CArray] Array that contains values that *param_name* will assumes during the attack (this define how the attacker power increases). If the first value is not zero, zero will be added as first value

save_adv_ds [bool, optional] If True, the samples at each parameter will be stored. Default False.

See also:

CAttack class that implements the attack.

Attributes

attack Return the attack object that is used from CSecEval to perform the attack.

class_type Defines class type.

logger Logger for current object.

save_adv_ds Returns

sec_eval_data Get a sec eval data objects.

verbose Verbosity level of logger output.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_data(self, path)	Restore Sec Eval data from file.
load_state(self, path)	Sets the object state from file.
run_sec_eval(self, dataset, **kwargs)	Performs attack while the power of the attacker (named param_name) increase.
save(self, path)	Save class object to file.
save_data(self, path)	Store Sec Eval data to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property attack

Return the attack object that is used from CSecEval to perform the attack.

load_data (*self*, *path*)

Restore Sec Eval data from file.

run_sec_eval (*self*, *dataset*, ***kwargs*)

Performs attack while the power of the attacker (named param_name) increase.

Parameters

dataset [CDataset] Dataset that contain samples that will be manipulated from the attacker while his attack power increase

kwargs Additional keyword arguments for the *CAttack.run* method.

property save_adv_ds

Returns

True/False: whether to store or not the manipulated attack sample dataset

save_data(*self*, *path*)
Store Sec Eval data to file.

property sec_eval_data
Get a sec eval data objects. It contains the Security Evaluation Results.

Returns

sec_eval_data: CSecEvalData object contains classifier security evaluation results

CSecEvalData

class secml.adv.seceval.c_sec_eval_data.**CSecEvalData**
Bases: *secml.core.c_creator.CCreator*

This class is a container for data computed during Classifier Security Evaluation.

Attributes

class_type ['standard'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Load Security evaluation data from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Load Security evaluation data from file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property Y
Returns the values of the dataset true labels.

property Y_pred

Returns

Y_pred [list of CArray] Contain one element for each attack power value. Each element contain label assigned to all the dataset samples from the attack.

property Y_target
Returns the values of the desired predicted labels.

property adv_ds

Returns

adv_ds [list of CDataset.] containing one dataset for each different parameter value.

property fobj
Return objective function values with the different attack power

classmethod load(*path*)
Load Security evaluation data from file.

Save a python dict containing all the results.

property param_name
Returns the name of the parameter representing the attack strenght.

property param_values
Returns the values of the security-evaluation parameter.

save(*self*, *path*)
Load Security evaluation data from file.

Save a python dict containing all the results.

property scores

Returns

scores: list of CArray Contain one element for each attack power value. Each element contain score assigned by the classifier to all the dataset samples.

property time

Returns

time [CArray (n_patterns, num parameter values)] Each array row contain the times of the attack for one samples. Each row element represent a different attack power.

4.7.9 secml.optim

Optimization

secml.optim.function

CFunction

class secml.optim.function.c_function.**CFunction**(*fun=None*, *gradient=None*, *n_dim=None*)

Bases: *secml.core.c_creator.CCreator*

Class that handles generic mathematical functions.

Either a function or its gradient can be passed in.

Number of expected space dimensions can be specified if applicable.

Parameters

fun [callable or None] Any python callable. Required if *gradient* is None.

gradient [callable or None] Any python callable that returns the gradient of *fun*. Required if *fun* is None.

n_dim [int or None, optional] Expected space dimensions.

Attributes

class_type ['generic'] Defines class type.

Methods

approx_fprime(self, x, epsilon, *args, **kwargs)	Finite-difference approximation of the gradient of a scalar function.
check_grad(self, x, epsilon, *args, **kwargs)	Check the correctness of a gradient function by comparing
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
fun(self, x, *args, **kwargs)	Evaluates function on x.
fun_ndarray(self, x, *args, **kwargs)	Evaluates function on x (ndarray).
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
gradient(self, x, *args, **kwargs)	Evaluates gradient of function at point x.
gradient_ndarray(self, x, *args, **kwargs)	Evaluates gradient of function at point x (ndarray).
has_fun(self)	True if function has been set.
has_gradient(self)	True if gradient has been set.
is_equal(self, x, val[, tol])	Evaluates if function value is close to *val* within tol.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
reset_eval(self)	Reset the count of function and gradient of function evaluations.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

approx_fprime(*self*, *x*, *epsilon*, **args*, ***kwargs*)

Finite-difference approximation of the gradient of a scalar function.

Wrapper for scipy function scipy.optimize.approx_fprime.

Parameters

x [CArray] The flat dense vector with the point at which to determine the gradient of *fun*.

epsilon [scalar or CArray] Increment of *x* to use for determining the function gradient. If a scalar, uses the same finite difference delta for all partial derivatives. If an array, should contain one value per element of *x*.

args, kwargs Any other arguments that are to be passed to *fun*.

Returns

grad [CArray] The gradient of *fun* at *x*.

See also:

check_grad Check correctness of function gradient against *approx_fprime*.

Notes

The function gradient is determined by the forward finite difference formula:

```
             fun(xk[i] + epsilon[i]) - f(xk[i])
fun'[i] = -----------------------------------
                        epsilon[i]
```

The main use of *approx_fprime* is to determine numerically the Jacobian of a function.

Examples

```
>>> from secml.array import CArray
>>> from secml.optim.function import CFunction
>>> from secml.core.constants import eps
```

```
>>> def func(x, c0, c1):
...     "Coordinate vector `x` should be an array of size two."
...     return c0 * x[0]**2 + c1*x[1]**2
```

```
>>> c0, c1 = (1, 200)
>>> CFunction(func).approx_fprime(CArray.ones(2), [eps, (200 ** 0.5) * eps],␣
↪c0, c1=c1)
CArray(2,)(dense: [  2.       400.000042])
```

check_grad(*self*, *x*, *epsilon*, **args*, ***kwargs*)

Check the correctness of a gradient function by comparing it against a (forward) finite-difference approximation of the gradient.

Parameters

x [CArray] Flat dense pattern to check function gradient against forward difference approximation of function gradient.

epsilon [scalar or CArray] Increment of *x* to use for determining the function gradient. If a scalar, uses the same finite difference delta for all partial derivatives. If an array, should contain one value per element of *x*.

args, kwargs Extra arguments passed to *fun* and *fprime*.

Returns

err [float] The square root of the sum of squares (i.e. the l2-norm) of the difference between `fprime(x, *args)` and the finite difference approximation of *fprime* at the points *x*.

See also:

approx_fprime Finite-difference approximation of the gradient of a scalar function.

Notes

epsilon is the only keyword argument accepted by the function. Any other optional argument for *fun* and *fprime* should be passed as non-keyword.

Examples

```
>>> from secml.optim.function import CFunction
>>> from secml.array import CArray
```

```
>>> def func(x):
...     return x[0].item()**2 - 0.5 * x[1].item()**3
>>> def grad(x):
...     return CArray([2 * x[0].item(), -1.5 * x[1].item()**2])
```

```
>>> fun = CFunction(func, grad)
>>> fun.check_grad(CArray([1.5, -1.5]), epsilon=1e-8)
7.817837928307533e-08
```

fun (*self*, *x*, **args*, ***kwargs*)
Evaluates function on x.

Parameters

x [CArray] Argument of fun.

args, kwargs Other optional parameter of the function.

Returns

out_fun [scalar or CArray] Function output, scalar or CArray depending on the inner function.

fun_ndarray (*self*, *x*, **args*, ***kwargs*)
Evaluates function on x (ndarray).

Parameters

x [np.ndarray] Argument of fun as ndarray.

args, kwargs Other optional parameter of the function.

Returns

out_fun [scalar or CArray] Function output, scalar or CArray depending on the inner function.

gradient (*self*, *x*, **args*, ***kwargs*)
Evaluates gradient of function at point x.

Parameters

x [CArray] Argument of gradient. Single point.

args, kwargs Other optional parameter of the function.

Returns

out_grad [CArray] Array with gradient output.

gradient_ndarray (*self*, *x*, **args*, ***kwargs*)
Evaluates gradient of function at point x (ndarray).

Parameters

x [ndarray] Argument of gradient.

args, kwargs Other optional parameter of the function.

Returns

out_grad [ndarray] Array with gradient output.

has_fun (*self*)
True if function has been set.

has_gradient (*self*)
True if gradient has been set.

is_equal (*self*, *x*, *val*, *tol=1e-06*)
Evaluates if function value is close to *val* within tol.

property n_dim
Returns the expected function's space dimensions.

property n_fun_eval
Returns the number of function evaluations.

property n_grad_eval
Returns the number of gradient evaluations.

reset_eval (*self*)
Reset the count of function and gradient of function evaluations.

CFunctionLinear

class secml.optim.function.c_function_linear.**CFunctionLinear**(*b*, *c*)
Bases: *secml.optim.function.c_function.CFunction*

Implements linear functions of the form: b' x + c = 0

Attributes

class_type ['linear'] Defines class type.

Methods

approx_fprime(self, x, epsilon, *args, **kwargs)	Finite-difference approximation of the gradient of a scalar function.
check_grad(self, x, epsilon, *args, **kwargs)	Check the correctness of a gradient function by comparing
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
fun(self, x, *args, **kwargs)	Evaluates function on x.
fun_ndarray(self, x, *args, **kwargs)	Evaluates function on x (ndarray).
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.

Continued on next page

Table 131 – continued from previous page

get_subclasses()	Get all the subclasses of the calling class.
gradient(self, x, *args, **kwargs)	Evaluates gradient of function at point x.
gradient_ndarray(self, x, *args, **kwargs)	Evaluates gradient of function at point x (ndarray).
has_fun(self)	True if function has been set.
has_gradient(self)	True if gradient has been set.
is_equal(self, x, val[, tol])	Evaluates if function value is close to *val* within tol.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
reset_eval(self)	Reset the count of function and gradient of function evaluations.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

CFunctionQuadratic

class secml.optim.function.c_function_quadratic.**CFunctionQuadratic**(*A*, *b*, *c*)

Bases: *secml.optim.function.c_function.CFunction*

Implements quadratic functions of the form: x' A x + b' x + c = 0

Attributes

class_type ['quadratic'] Defines class type.

Methods

approx_fprime(self, x, epsilon, *args, **kwargs)	Finite-difference approximation of the gradient of a scalar function.
check_grad(self, x, epsilon, *args, **kwargs)	Check the correctness of a gradient function by comparing
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
fun(self, x, *args, **kwargs)	Evaluates function on x.
fun_ndarray(self, x, *args, **kwargs)	Evaluates function on x (ndarray).
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
gradient(self, x, *args, **kwargs)	Evaluates gradient of function at point x.
gradient_ndarray(self, x, *args, **kwargs)	Evaluates gradient of function at point x (ndarray).

Continued on next page

Table 132 – continued from previous page

has_fun(self)	True if function has been set.
has_gradient(self)	True if gradient has been set.
is_equal(self, x, val[, tol])	Evaluates if function value is close to *val* within tol.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
reset_eval(self)	Reset the count of function and gradient of function evaluations.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

CFunctionRosenbrock

class secml.optim.function.c_function_rosenbrock.**CFunctionRosenbrock**

Bases: *secml.optim.function.c_function.CFunction*

The Rosenbrock function.

Non-convex function introduced by Howard H. Rosenbrock in 1960. [1] Also known as Rosenbrock's valley or Rosenbrock's banana function.

Global minimum f(x) = 0 @ x = (1, 1, . . ., 1).

Given by: .. math:

```
f(x) = \sum^{n-1}_{i=1} [100 * {(x_{i+1} - x_i^2)}^2 + (x_i - 1)^2]
```

References

[1]

Attributes

class_type ['rosenbrock'] Defines class type.

Methods

approx_fprime(self, x, epsilon, *args, **kwargs)	Finite-difference approximation of the gradient of a scalar function.
check_grad(self, x, epsilon, *args, **kwargs)	Check the correctness of a gradient function by comparing
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.

Continued on next page

Table 133 – continued from previous page

fun(self, x, *args, **kwargs)	Evaluates function on x.
fun_ndarray(self, x, *args, **kwargs)	Evaluates function on x (ndarray).
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
global_min()	Value of the global minimum of the function.
global_min_x([ndim])	Global minimum point of the function.
gradient(self, x, *args, **kwargs)	Evaluates gradient of function at point x.
gradient_ndarray(self, x, *args, **kwargs)	Evaluates gradient of function at point x (ndarray).
has_fun(self)	True if function has been set.
has_gradient(self)	True if gradient has been set.
is_equal(self, x, val[, tol])	Evaluates if function value is close to *val* within tol.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
reset_eval(self)	Reset the count of function and gradient of function evaluations.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

static global_min()

Value of the global minimum of the function.

Global minimum f(x) = 0 @ x = (1, 1, ..., 1).

Returns

float Value of the global minimum of the function.

static global_min_x(*ndim=2*)

Global minimum point of the function.

Global minimum f(x) = 0 @ x = (1, 1, ..., 1).

Parameters

ndim [int, optional] Number of dimensions to consider, higher or equal to 2. Default 2.

Returns

CArray The global minimum point of the function.

CFunctionThreeHumpCamel

class secml.optim.function.c_function_3hcamel.**CFunctionThreeHumpCamel**

Bases: *secml.optim.function.c_function.CFunction*

The Three-Hump Camel function.

2-Dimensional function.

Global minimum f(x) = 0 @ x = (0, 0).

Given by: .. math:

```
f(x) = 2 * x_0 ** 2 - 1.05 * x_0 ** 4 +
 x_0 ** 6 / 6 + x_0 * x_1 + x_1 ^ 2
```

Attributes

class_type ['3h-camel'] Defines class type.

Methods

approx_fprime(self, x, epsilon, *args, **kwargs)	Finite-difference approximation of the gradient of a scalar function.
check_grad(self, x, epsilon, *args, **kwargs)	Check the correctness of a gradient function by comparing
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
fun(self, x, *args, **kwargs)	Evaluates function on x.
fun_ndarray(self, x, *args, **kwargs)	Evaluates function on x (ndarray).
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
global_min()	Value of the global minimum of the function.
global_min_x()	Global minimum point of the function.
gradient(self, x, *args, **kwargs)	Evaluates gradient of function at point x.
gradient_ndarray(self, x, *args, **kwargs)	Evaluates gradient of function at point x (ndarray).
has_fun(self)	True if function has been set.
has_gradient(self)	True if gradient has been set.
is_equal(self, x, val[, tol])	Evaluates if function value is close to *val* within tol.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
reset_eval(self)	Reset the count of function and gradient of function evaluations.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.

Continued on next page

Table 134 – continued from previous page

set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

static global_min()
Value of the global minimum of the function.

Global minimum f(x) = 0 @ x = (0, 0).

Returns

float Value of the global minimum of the function.

static global_min_x()
Global minimum point of the function.

Global minimum f(x) = 0 @ x = (0, 0).

Returns

CArray The global minimum point of the function.

CFunctionBeale

class secml.optim.function.c_function_beale.**CFunctionBeale**
Bases: *secml.optim.function.c_function.CFunction*

The Beale function.

2-Dimensional, multimodal, with sharp peaks at the corners of the input domain.

Global minimum f(x) = 0 @ x = (3, 0.5).

Given by: .. math:

```
f(x) = (1.5 - x_0 + x_0 * x_1)^2 + (2.25 - x_0 + x_0 * x_1^2)^2 +
 (2.625 - x_0 + x_0 * x_1^3)^2
```

Attributes

class_type ['beale'] Defines class type.

Methods

approx_fprime(self, x, epsilon, *args, **kwargs)	Finite-difference approximation of the gradient of a scalar function.
check_grad(self, x, epsilon, *args, **kwargs)	Check the correctness of a gradient function by comparing
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
fun(self, x, *args, **kwargs)	Evaluates function on x.
fun_ndarray(self, x, *args, **kwargs)	Evaluates function on x (ndarray).

Continued on next page

Table 135 – continued from previous page

get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
global_min()	Value of the global minimum of the function.
global_min_x()	Global minimum point of the function.
gradient(self, x, *args, **kwargs)	Evaluates gradient of function at point x.
gradient_ndarray(self, x, *args, **kwargs)	Evaluates gradient of function at point x (ndarray).
has_fun(self)	True if function has been set.
has_gradient(self)	True if gradient has been set.
is_equal(self, x, val[, tol])	Evaluates if function value is close to *val* within tol.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
reset_eval(self)	Reset the count of function and gradient of function evaluations.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

static global_min()

Value of the global minimum of the function.

Global minimum f(x) = 0 @ x = (3, 0.5).

Returns

float Value of the global minimum of the function.

static global_min_x()

Global minimum point of the function.

Global minimum f(x) = 0 @ x = (3, 0.5).

Returns

CArray The global minimum point of the function.

CFunctionMcCormick

class secml.optim.function.c_function_mccormick.**CFunctionMcCormick**

Bases: *secml.optim.function.c_function.CFunction*

The McCormick function.

2-Dimensional function.

Global minimum f(x) = -1.9132 @ x = (-0.5472, -1.5472). This is on a compact domain (lb=[-1.5,-3], ub=[4,4])

Given by: .. math:

```
f(x) = sin(x_0 + x_1) + (x_0 - x_1)^2 - 1.5 * x_0 + 2.5 * x_1 + 1
```

Attributes

class_type [mc-cormick'] Defines class type.

Methods

approx_fprime(self, x, epsilon, *args, **kwargs)	Finite-difference approximation of the gradient of a scalar function.
check_grad(self, x, epsilon, *args, **kwargs)	Check the correctness of a gradient function by comparing
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
fun(self, x, *args, **kwargs)	Evaluates function on x.
fun_ndarray(self, x, *args, **kwargs)	Evaluates function on x (ndarray).
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
global_min()	Value of the global minimum of the function.
global_min_x()	Global minimum point of the function.
gradient(self, x, *args, **kwargs)	Evaluates gradient of function at point x.
gradient_ndarray(self, x, *args, **kwargs)	Evaluates gradient of function at point x (ndarray).
has_fun(self)	True if function has been set.
has_gradient(self)	True if gradient has been set.
is_equal(self, x, val[, tol])	Evaluates if function value is close to *val* within tol.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
reset_eval(self)	Reset the count of function and gradient of function evaluations.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

bounds

static bounds()

static global_min()

Value of the global minimum of the function.

Global minimum f(x) = -1.9132 @ x = (-0.5472, -1.5472).

Returns

float Value of the global minimum of the function.

static global_min_x()
Global minimum point of the function.

Global minimum f(x) = -1.9132 @ x = (-0.5472, -1.5472).

Returns

CArray The global minimum point of the function.

secml.optim.optimizers

secml.optim.optimizers.line_search

CLineSearch

class secml.optim.optimizers.line_search.c_line_search.**CLineSearch**(*fun*, *constr=None*, *bounds=None*, *eta=0.0001*, *max_iter=20*)

Bases: *secml.core.c_creator.CCreator*

Abstract class that implements line-search optimization algorithms.

Line-search algorithms optimize the objective function along a given direction in the feasible domain, potentially subject to constraints. The search is normally stopped when the objective improves at a satisfying level, to keep the search fast.

Attributes

class_type Defines class type.

logger Logger for current object.

verbose Verbosity level of logger output.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
minimize(self, x, d, **kwargs)	Line search.

Continued on next page

Table 137 – continued from previous page

save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

abstract minimize(*self*, *x*, *d*, ***kwargs*)

Line search.

Parameters

x [CArray] The input point.

d [CArray] The descent direction along which fun(x) is minimized.

kwargs [dict] Additional parameters required to evaluate *fun(x, **kwargs)*.

CLineSearchBisect

class secml.optim.optimizers.line_search.c_line_search_bisect.**CLineSearchBisect**(*fun*, *constr=None*, *bounds=None*, *eta=0.0001*, *eta_min=0.1*, *eta_max=None*, *max_iter=20*)

Bases: *secml.optim.optimizers.line_search.c_line_search.CLineSearch*

Binary line search.

Attributes

class_type ['bisect'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
minimize(self, x, d[, fx, tol])	Bisect line search (on discrete grid).

Continued on next page

Table 138 – continued from previous page

save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property eta_max

property eta_min

minimize(*self*, *x*, *d*, *fx=None*, *tol=0.0001*, ***kwargs*)

Bisect line search (on discrete grid).

The function *fun(x + a*eta*d)* with *a = {0, 1, 2, . . . }* is minimized along the descent direction d.

If *fun(x) >= 0* -> step_min = step else step_max = step

If eta_max is not None, it runs a bisect line search in *[x + eta_min*d, x + eta_max*d]; otherwise, it runs an exponential line search in `[x + eta*d, . . . , x + eta_min*d, . . .]*

Parameters

x [CArray] The input point.

d [CArray] The descent direction along which *fun(x)* is minimized.

fx [int or float or None, optional] The current value of *fun(x)* (if available).

tol [float, optional] Tolerance for convergence to the local minimum.

kwargs [dict] Additional parameters required to evaluate *fun(x, **kwargs)*.

Returns

x' [CArray] Point *x' = x + eta * d* that approximately solves *min f(x + eta*d)*.

fx': int or float or None, optional The value *f(x')*.

property n_iter

CLineSearchBisectProj

class secml.optim.optimizers.line_search.c_line_search_bisect_proj.**CLineSearchBisectProj**(*fu*
co
st
bo
et
et
et
m

Bases: *secml.optim.optimizers.line_search.c_line_search_bisect.CLineSearchBisect*

Binary line search including projections.

Attributes

class_type ['bisect-proj'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
minimize(self, x, d[, fx, tol])	Exponential line search (on discrete grid).
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

minimize (*self*, *x*, *d*, *fx=None*, *tol=0.0001*, ***kwargs*)

Exponential line search (on discrete grid).

The function *fun(x + a*eta*d)* with *a = {0, 1, 2, . . . }* is minimized along the descent direction d.

If *fun(x) >= 0* -> step_min = step else step_max = step

If eta_max is not None, it runs a bisect line search in *[x + eta_min*d, x + eta_max*d]; otherwise, it runs an exponential line search in `[x + eta*d, . . . , x + eta_min*d, . . .]*

Parameters

x [CArray] The input point.

d [CArray] The descent direction along which *fun(x)* is minimized.

fx [int or float or None, optional] The current value of *fun(x)* (if available).

tol [float, optional] Tolerance for convergence to the local minimum.

kwargs [dict] Additional parameters required to evaluate *fun(x, **kwargs)*.

Returns

x' [CArray] Point *x' = x + eta * d* that approximately solves *min f(x + eta*d)*.

fx': int or float or None, optional The value *f(x')*.

COptimizer

class secml.optim.optimizers.c_optimizer.**COptimizer**(*fun*, *constr=None*, *bounds=None*)

Bases: *secml.core.c_creator.CCreator*

Interface for optimizers.

Implements: minimize f(x) s.t. gi(x) <= 0, i=1,...,m (inequality constraints) hj(x) = 0, j = 1,..., n (equality constraints)

Parameters

fun [CFunction] The objective function to be optimized, along with 1st-order (Jacobian) and 2nd-order (Hessian) derivatives (if available).

Attributes

bounds Optimization bounds.

class_type Defines class type.

constr Optimization constraint.

f The objective function

f_eval

f_opt

f_seq

grad_eval

logger Logger for current object.

n_dim

verbose Verbosity level of logger output.

x_opt

x_seq

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
maximize(self, x_init[, args])	Interface for maximizers.

Continued on next page

Table 140 – continued from previous page

minimize(self, x_init[, args])	Interface for minimizers.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property bounds
Optimization bounds.

property constr
Optimization constraint.

property f
The objective function

property f_eval

property f_opt

property f_seq

property grad_eval

maximize(*self*, *x_init*, *args=()*, ***kwargs*)
Interface for maximizers.

Implementing: max fun(x) s.t. constraint

This is implemented by inverting the sign of fun and gradient and running the *COptimizer.minimize()*.

Parameters

x_init [CArray] The initial input point.

args [tuple, optional] Extra arguments passed to the objective function and its gradient.

kwargs Additional parameters of the minimization method.

abstract minimize(*self*, *x_init*, *args=()*, ***kwargs*)
Interface for minimizers.

Implementing: min fun(x) s.t. constraint

Parameters

x_init [CArray] The initial input point.

args [tuple, optional] Extra arguments passed to the objective function and its gradient.

kwargs Additional parameters of the minimization method.

property n_dim

property x_opt

property x_seq

COptimizerPGD

class secml.optim.optimizers.c_optimizer_pgd.**COptimizerPGD**(*fun*, *constr=None*, *bounds=None*, *eta=0.001*, *eps=0.0001*, *max_iter=200*)

Bases: *secml.optim.optimizers.c_optimizer.COptimizer*

Solves the following problem:

min f(x) s.t. d(x,x0) <= dmax x_lb <= x <= x_ub

f(x) is the objective function (either linear or nonlinear), d(x,x0) <= dmax is a distance constraint in feature space (l1 or l2), and x_lb <= x <= x_ub is a box constraint on x.

The solution algorithm is based on the classic gradient descent algorithm.

Attributes

class_type ['pgd'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
maximize(self, x_init[, args])	Interface for maximizers.
minimize(self, x_init[, args])	Interface to minimizers.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property eps
Return tolerance value for stop criterion

property eta
Return gradient descent step

property max_iter
Returns the maximum number of gradient descent iteration

minimize(*self*, *x_init*, *args=()*, ***kwargs*)

Interface to minimizers.

Implements: min fun(x) s.t. constraint

Parameters

x_init [CArray] The initial input point.

args [tuple, optional] Extra arguments passed to the objective function and its gradient.

Returns

f_seq [CArray] Array containing values of f during optimization.

x_seq [CArray] Array containing values of x during optimization.

COptimizerPGDLS

class secml.optim.optimizers.c_optimizer_pgd_ls.**COptimizerPGDLS**(*fun*, *constr=None*, *bounds=None*, *discrete=False*, *eta=0.001*, *eta_min=None*, *eta_max=None*, *max_iter=1000*, *eps=0.0001*)

Bases: *secml.optim.optimizers.c_optimizer.COptimizer*

Solves the following problem:

min f(x) s.t. d(x,x0) <= dmax x_lb <= x <= x_ub

f(x) is the objective function (either linear or nonlinear), d(x,x0) <= dmax is a distance constraint in feature space (l1 or l2), and x_lb <= x <= x_ub is a box constraint on x.

The solution algorithm is based on a line-search exploring one feature (i.e., dimension) at a time (for l1-constrained problems), or all features (for l2-constrained problems). This solver also works for discrete problems, where x is integer valued. In this case, exploration works by manipulating one feature at a time.

Differently from standard line searches, it explores a subset of *n_dimensions* at a time. In this sense, it is an extension of the classical line-search approach.

Attributes

class_type ['pgd-ls'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.

Continued on next page

Table 142 – continued from previous page

get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
maximize(self, x_init[, args])	Interface for maximizers.
minimize(self, x_init[, args])	Interface to minimizers implementing
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property discrete
True if feature space is discrete, False if continuous.

property eps
Return tolerance value for stop criterion

property eta

property eta_max

property eta_min

property max_iter
Returns the maximum number of descent iterations

minimize(*self*, *x_init*, *args=()*, ***kwargs*)

Interface to minimizers implementing min fun(x) s.t. constraint

Parameters

x_init [CArray] The initial input point.

args [tuple, optional] Extra arguments passed to the objective function and its gradient.

Returns

f_seq [CArray] Array containing values of f during optimization.

x_seq [CArray] Array containing values of x during optimization.

COptimizerPGDExp

class secml.optim.optimizers.c_optimizer_pgd_exp.**COptimizerPGDExp**(*fun*, *constr=None*, *bounds=None*, *discrete=False*, *eta=0.001*, *eta_min=None*, *eta_max=None*, *max_iter=1000*, *eps=0.0001*)

Bases: *secml.optim.optimizers.c_optimizer.COptimizer*

Solves the following problem:

min f(x) s.t. d(x,x0) <= dmax x_lb <= x <= x_ub

f(x) is the objective function (either linear or nonlinear), d(x,x0) <= dmax is a distance constraint in feature space (l1 or l2), and x_lb <= x <= x_ub is a box constraint on x.

The solution algorithm is based on a line-search exploring one feature (i.e., dimension) at a time (for l1-constrained problems), or all features (for l2-constrained problems).

Attributes

class_type ['pgd-exp'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
maximize(self, x_init[, args])	Interface for maximizers.
minimize(self, x_init[, args])	Interface to minimizers implementing
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property discrete

True if feature space is discrete, False if continuous.

property eps
Return tolerance value for stop criterion

property eta

property eta_max

property eta_min

property max_iter
Returns the maximum number of descent iterations

minimize(*self*, *x_init*, *args=()*, ***kwargs*)

Interface to minimizers implementing min fun(x) s.t. constraint

Parameters

x_init [CArray] The initial input point.

args [tuple, optional] Extra arguments passed to the objective function and its gradient.

Returns

f_seq [CArray] Array containing values of f during optimization.

x_seq [CArray] Array containing values of x during optimization.

COptimizerScipy

class secml.optim.optimizers.c_optimizer_scipy.**COptimizerScipy**(*fun*, *constr=None*, *bounds=None*)

Bases: *secml.optim.optimizers.c_optimizer.COptimizer*

Implements optimizers from scipy.

Attributes

class_type ['scipy-opt'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
maximize(self, x_init[, args])	Interface for maximizers.
minimize(self, x_init[, args])	Minimize function.

Continued on next page

Table 144 – continued from previous page

save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

minimize (*self*, *x_init*, *args=()*, ***kwargs*)

Minimize function.

Wrapper of *scipy.optimize.minimize*.

Parameters

x_init [CArray] Init point. Dense flat array of real elements of size 'n', where 'n' is the number of independent variables.

args [tuple, optional] Extra arguments passed to the objective function and its derivatives (*fun*, *jac* and *hess* functions).

The following can be passed as optional keyword arguments:

method [str or callable, optional] Type of solver. Should be one of

- 'BFGS' (see here)
- 'L-BFGS-B' (see here)

If not given, chosen to be one of BFGS or L-BFGS-B depending if the problem has constraints or bounds. See *c_optimizer_scipy.SUPPORTED_METHODS* for the full list.

jac [{'2-point', '3-point', 'cs', bool}, optional] Method for computing the gradient vector. The function in *self.fun.gradient* will be used (if defined). Alternatively, the keywords {'2-point', '3-point', 'cs'} select a finite difference scheme for numerical estimation of the gradient. Options '3-point' and 'cs' are available only to 'trust-constr'. If *jac* is a Boolean and is True, *fun* is assumed to return the gradient along with the objective function. If False, the gradient will be estimated using '2-point' finite difference estimation.

bounds [scipy.optimize.Bounds, optional] A bound constraint in scipy.optimize format. If defined, bounds of *COptimizerScipy* will be ignored.

tol [float, optional] Tolerance for termination. For detailed control, use solver-specific options.

options [dict, optional] A dictionary of solver options. All methods accept the following generic options:

- maxiter : int Maximum number of iterations to perform.
- disp : bool Set to True to print convergence messages. Equivalent of setting *COptimizerScipy.verbose = 2*.

For method-specific options, see show_options.

Returns

x [CArray] The solution of the optimization.

Examples

```
>>> from secml.array import CArray
>>> from secml.optim.optimizers import COptimizerScipy
>>> from secml.optim.function import CFunctionRosenbrock
```

```
>>> x_init = CArray([1.3, 0.7])
>>> opt = COptimizerScipy(CFunctionRosenbrock())
>>> x_opt = opt.minimize(
... x_init, method='BFGS', options={'gtol': 1e-6, 'disp': True})
Optimization terminated successfully.
         Current function value: 0.000000
         Iterations: 32
         Function evaluations: 39
         Gradient evaluations: 39
>>> print(x_opt)
CArray([1. 1.])
>>> print(opt.f_opt)
9.294383981640425e-19
```

secml.optim.constraints

CConstraint

class secml.optim.constraints.c_constraint.**CConstraint**

Bases: *secml.core.c_creator.CCreator*

Interface for equality/inequality constraints.

Attributes

class_type Defines class type.

logger Logger for current object.

verbose Verbosity level of logger output.

Methods

constraint(self, x)	Returns the value of the constraint for the sample x.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
gradient(self, x)	Returns the gradient of c(x) in x.
is_active(self, x[, tol])	Returns True if constraint is active.
is_violated(self, x)	Returns the violated status of the constraint for the sample x.

Continued on next page

Table 145 – continued from previous page

list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
projection(self, x)	Project x onto feasible domain / within the given constraint.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

constraint (*self*, *x*)

Returns the value of the constraint for the sample x.

Parameters

x [CArray] Input sample.

Returns

float Value of the constraint.

gradient (*self*, *x*)

Returns the gradient of c(x) in x.

Parameters

x [CArray] Input sample.

Returns

CArray The gradient of the constraint computed on x.

is_active (*self*, *x*, *tol=0.0001*)

Returns True if constraint is active.

A constraint is active if c(x) = 0.

By default we assume constraints of the form c(x) <= 0.

Parameters

x [CArray] Input sample.

tol [float, optional] Tolerance to use for comparing c(x) against 0. Default 1e-4.

Returns

bool True if constraint is active, False otherwise.

is_violated (*self*, *x*)

Returns the violated status of the constraint for the sample x.

We assume the constraint violated if c(x) <= 0.

Parameters

x [CArray] Input sample.

Returns

bool True if constraint is violated, False otherwise.

projection(*self*, *x*)

Project x onto feasible domain / within the given constraint.

If constraint is not violated by x, x is returned.

Parameters

x [CArray] Input sample.

Returns

CArray Projected x onto feasible domain if constraint is violated. Otherwise, x is returned as is.

CConstraintBox

class secml.optim.constraints.c_constraint_box.**CConstraintBox**(*lb=None*, *ub=None*)

Bases: *secml.optim.constraints.c_constraint.CConstraint*

Class that defines a box constraint.

Parameters

lb, ub [scalar or CArray or None, optional] Bounds of the constraints. If scalar, the same bound will be applied to all features. If CArray, should contain a bound for each feature. If None, a +/- inf ub/lb bound will be used for all features.

Attributes

class_type ['box'] Defines class type.

Methods

constraint(self, x)	Returns the value of the constraint for the sample x.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
gradient(self, x)	Returns the gradient of c(x) in x.
is_active(self, x[, tol])	Returns True if constraint is active.
is_violated(self, x)	Returns the violated status of the constraint for the sample x.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
projection(self, x)	Project x onto feasible domain / within the given constraint.
save(self, path)	Save class object to file.

Continued on next page

Table 146 – continued from previous page

save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_center_radius(self, c, r)	Set constraint bounds in terms of center and radius.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property center
Center of the constraint.

is_active(*self*, *x*, *tol=0.0001*)
Returns True if constraint is active.

A constraint is active if c(x) = 0.

By default we assume constraints of the form c(x) <= 0.

Parameters

x [CArray] Input sample.

tol [float, optional] Tolerance to use for comparing c(x) against 0. Default 1e-4.

Returns

bool True if constraint is active, False otherwise.

is_violated(*self*, *x*)
Returns the violated status of the constraint for the sample x.

We assume the constraint violated if c(x) <= 0.

Parameters

x [CArray] Input sample.

Returns

bool True if constraint is violated, False otherwise.

property lb
Lower bound.

property radius
Radius of the constraint.

set_center_radius(*self*, *c*, *r*)
Set constraint bounds in terms of center and radius.

This method will transform the input center/radius as follows: lb = center - radius ub = center + radius

Parameters

c [scalar] Constraint center.

r [scalar] Constraint radius.

property ub
Upper bound.

CConstraintL1

class secml.optim.constraints.c_constraint_l1.**CConstraintL1**(*center=0*, *radius=1*)

Bases: *secml.optim.constraints.c_constraint.CConstraint*

L1 Constraint.

Parameters

center [scalar or CArray, optional] Center of the constraint. Use an array to specify a different value for each dimension. Default 0.

radius [scalar, optional] The semidiagonal of the constraint. Default 1.

Attributes

class_type ['l1'] Defines class type.

Methods

constraint(self, x)	Returns the value of the constraint for the sample x.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
gradient(self, x)	Returns the gradient of c(x) in x.
is_active(self, x[, tol])	Returns True if constraint is active.
is_violated(self, x)	Returns the violated status of the constraint for the sample x.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
projection(self, x)	Project x onto feasible domain / within the given constraint.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property center

Center of the constraint.

property radius

Semidiagonal of the constraint.

CConstraintL2

class secml.optim.constraints.c_constraint_l2.**CConstraintL2**(*center=0*, *radius=1*)

Bases: *secml.optim.constraints.c_constraint.CConstraint*

L2 Constraint.

Parameters

center [scalar or CArray, optional] Center of the constraint. Use an array to specify a different value for each dimension. Default 0.

radius [scalar, optional] The semidiagonal of the constraint. Default 1.

Attributes

class_type ['l2'] Defines class type.

Methods

constraint(self, x)	Returns the value of the constraint for the sample x.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
gradient(self, x)	Returns the gradient of c(x) in x.
is_active(self, x[, tol])	Returns True if constraint is active.
is_violated(self, x)	Returns the violated status of the constraint for the sample x.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
projection(self, x)	Project x onto feasible domain / within the given constraint.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property center

Center of the constraint.

property radius

Radius of the constraint.

4.7.10 secml.model_zoo

load_model

secml.model_zoo.load_model.**load_model**(*model_id*)

Load a pre-trained classifier.

Returns a pre-trained SecML classifier given the id of the model.

Check https://gitlab.com/secml/secml-zoo for the list of available models.

Parameters

model_id [str] Identifier of the pre-trained model to load.

Returns

CClassifier Desired pre-trained model.

4.7.11 secml.explanation

CExplainer

class secml.explanation.c_explainer.**CExplainer**(*clf*)

Bases: *secml.core.c_creator.CCreator*

Abstract interface for Explainable ML methods.

Parameters

clf [CClassifier] Instance of the classifier to explain.

Attributes

class_type Defines class type.

clf Classifier to explain.

logger Logger for current object.

verbose Verbosity level of logger output.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
explain(self, x, *args, **kwargs)	Computes the explanation on x.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.

Continued on next page

Table 149 – continued from previous page

save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

property clf
Classifier to explain.

abstract explain(*self*, *x*, **args*, ***kwargs*)
Computes the explanation on x.

CExplainerGradient

class secml.explanation.c_explainer_gradient.**CExplainerGradient**(*clf*)
Bases: *secml.explanation.c_explainer.CExplainer*

Explanation of predictions via input gradient.

The relevance *rv* of each feature is given by:

$$rv_i = \frac{\partial F(x)}{\partial x_i}$$

- D. Baehrens, T. Schroeter, S. Harmeling, M. Kawanabe, K. Hansen, K.-R.Muller, " "How to explain individual classification decisions", in: J. Mach. Learn. Res. 11 (2010) 1803-1831

Parameters

clf [CClassifier] Instance of the classifier to explain. Must be differentiable.

Attributes

class_type ['gradient'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
explain(self, x, y[, return_grad])	Computes the explanation for input sample.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.

Continued on next page

Table 150 – continued from previous page

save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

explain (*self*, *x*, *y*, *return_grad=False*)

Computes the explanation for input sample.

Parameters

x [CArray] Input sample.

y [int] Class wrt compute the classifier gradient.

return_grad [bool, optional] If True, also return the clf gradient computed on x. Default False.

Returns

relevance [CArray] Relevance vector for input sample.

CExplainerGradientInput

class secml.explanation.c_explainer_gradient_input.**CExplainerGradientInput**(*clf*)

Bases: *secml.explanation.c_explainer_gradient.CExplainerGradient*

Explanation of predictions via gradient*input vector.

The relevance *rv* of each features is given by:

$$rv_i(x) = \left(x_i * \frac{\partial F(x)}{\partial x_i} \right)$$

- A. Shrikumar, P. Greenside, A. Shcherbina, A. Kundaje, "Not just a blackbox: Learning important features through propagating activation differences", 2016 arXiv:1605.01713.
- M. Melis, D. Maiorca, B. Biggio, G. Giacinto and F. Roli, "Explaining Black-box Android Malware Detection," 2018 26th European Signal Processing Conference (EUSIPCO), Rome, 2018, pp. 524-528.

Parameters

clf [CClassifier] Instance of the classifier to explain. Must be differentiable.

Attributes

class_type ['gradient-input'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
explain(self, x, y[, return_grad])	Computes the explanation for input sample.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

explain (*self*, *x*, *y*, *return_grad=False*)

Computes the explanation for input sample.

Parameters

x [CArray] Input sample.

y [int] Class wrt compute the classifier gradient.

return_grad [bool, optional] If True, also return the clf gradient computed on x. Default False.

Returns

relevance [CArray] Relevance vector for input sample.

CExplainerIntegratedGradients

class secml.explanation.c_explainer_integrated_gradients.**CExplainerIntegratedGradients**(*clf*)

Bases: *secml.explanation.c_explainer_gradient.CExplainerGradient*

Explanation of predictions via integrated gradients.

This implements a method for local explanation of predictions via attribution of relevance to each feature.

The algorithm takes a sample and computes the Riemman approximation of the integral along the linear interpolation with a reference point.

- Sundararajan, Mukund, Ankur Taly, and Qiqi Yan. "Axiomatic Attribution for Deep Networks." Proceedings of the 34th International Conference on Machine Learning, Volume 70, JMLR. org, 2017, pp. 3319-3328.

So we have for each dimension *i* of the input sample x:

$$IG_i(x) = (x_i - x'_i) \times \sum_{k=1}^{m} \frac{\partial F(x' + \frac{k}{m} \times (x - x'))}{\partial x_i} \times \frac{1}{m}$$

with *m* the number of steps in the Riemman approximation of the integral.

Parameters

clf [CClassifier] Instance of the classifier to explain. Must be differentiable.

Attributes

class_type ['integrated-gradients'] Defines class type.

Methods

check_attributions(self, x, reference, c, ...)	Check proposition 1 on attributions.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
explain(self, x, y[, return_grad, reference, m])	Computes the explanation for input sample.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
linearly_interpolate(x[, reference, m])	Computes the linear interpolation between the sample and the reference.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

check_attributions(*self*, *x*, *reference*, *c*, *attributions*)

Check proposition 1 on attributions.

Proposition 1: Attributions should add up to the difference between the score at the input and that at the reference point.

Parameters

x [CArray] Input sample.

reference [CArray] The reference sample. Must have the same shape of input sample.

c [int] Class wrt the attributions have been computed.

attributions [CArray] Attributions for sample *x* to check.

explain (*self*, *x*, *y*, *return_grad=<no value>*, *reference=None*, *m=50*)
Computes the explanation for input sample.

Parameters

x [CArray] Input sample.

y [int] Class wrt compute the classifier gradient.

reference [CArray or None, optional] The reference sample. Must have the same shape of input sample. If None, a all-zeros sample will be used.

m [int, optional] The number of steps for linear interpolation. Default 50.+

Returns

attributions [CArray] Attributions (weight of each feature) for input sample.

static linearly_interpolate (*x*, *reference=None*, *m=50*)
Computes the linear interpolation between the sample and the reference.

Parameters

x [CArray] Input sample.

reference [CArray or None, optional] The reference sample. Must have the same shape of input sample. If None, a all-zeros sample will be used.

m [int, optional] The number of steps for linear interpolation. Default 50.

Returns

list List of CArrays to integrate over.

CExplainerInfluenceFunctions

class secml.explanation.c_explainer_influence_functions.**CExplainerInfluenceFunctions** (*clf*, *tr_ds*, *outer_lo*

Bases: *secml.explanation.c_explainer_gradient.CExplainerGradient*

Explanation of predictions via influence functions.

- Koh, Pang Wei, and Percy Liang, "Understanding black-box predictions via influence functions", in: Proceedings of the 34th International Conference on Machine Learning-Volume 70. JMLR. org, 2017.

Parameters

clf [CClassifier] Instance of the classifier to explain. Must provide the *hessian*.

tr_ds [CDataset] Training dataset of the classifier to explain.

Attributes

class_type ['influence-functions'] Defines class type.

Methods

copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
explain(self, x, y[, return_grad])	Compute influence of test sample x against all training samples.
get_class_from_type(class_type)	Return the class associated with input type.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
grad_inner_loss_params(self, x, y)	Compute derivative of the inner training loss function for all training points.
grad_outer_loss_params(self, x, y)	Compute derivate of the outer validation loss at test point(s) x This is typically not regularized (just an empirical loss function)
hessian(self, x, y)	Compute hessian for the current parameters of the trained clf.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
timed([msg])	Timer decorator.

explain (*self*, *x*, *y*, *return_grad=False*)
Compute influence of test sample x against all training samples.

Parameters

x [CArray] Input sample.

y [int] Class wrt compute the classifier gradient.

return_grad [bool, optional] If True, also return the clf gradient computed on x. Default False.

grad_inner_loss_params (*self*, *x*, *y*)
Compute derivative of the inner training loss function for all training points. This is normally a regularized loss.

grad_outer_loss_params (*self*, *x*, *y*)
Compute derivate of the outer validation loss at test point(s) x This is typically not regularized (just an empirical loss function)

hessian (*self*, *x*, *y*)
Compute hessian for the current parameters of the trained clf.

property tr_ds
Training dataset.

4.7.12 secml.figure

CFigure

class secml.figure.c_figure.**CFigure**(*height=6*, *width=6*, *title=''*, *fontsize=12*, *linewidth=2*, *markersize=7*)

Bases: *secml.core.c_creator.CCreator*

Creates a Figure.

A Figure is a collection of subplots. The last active subplot can be accessed by calling CFigure.sp`, followed by the desired plotting function (plot, scatter, contour, etc.). Each subplot is identified by an index (grid slot) inside an imaginary grid (n_rows, n_columns, grid_slot), counting from left to right, from top to bottom. By default, a subplot is created in a single-row, single-column imaginary grid (1, 1, 1).

Parameters

height [scalar, optional] Height of the new figure. Default 6.

width [scalar, optional] Width of the new figure. Default 6.

title [str, optional] Super title of the new figure. This is not the subplot title. To set a title for active subplot use `subtitle`. Default is to not set a super title.

linewidth [float, optional] Define default linewidth for all subplots. Default 2.

fontsize [int, optional] Define default fontsize for all subplots. Default 12.

markersize [scalar, optional] Define default markersize for all subplots. Default 7.

Notes

Not all *matplotlib* methods and/or parameters are currently available. If needed, directly access the *matplotlib.Axes* active subplot instance through the *CFigure.sp._sp* parameter.

Examples

```
>>> from secml.figure import CFigure
```

```
>>> fig = CFigure(fontsize=14)
>>> fig.sp.plot([0, 1], color='red')
```

```
>>> fig.show()  # This will open a new window with the figure
```

Attributes

class_type Defines class type.

logger Logger for current object.

n_sp Returns the number of created subplots.

sp Return reference to active subplot class.

verbose Verbosity level of logger output.

Methods

close(self[, fig])	Close current or input figure.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
get_class_from_type(class_type)	Return the class associated with input type.
get_default_params(self)	Return current defaults for the figure.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
savefig(self, fname[, dpi, facecolor, ...])	Save figure to disk.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
show([block])	Show all created figures.
subplot(self[, n_rows, n_cols, grid_slot])	Create a new subplot into specific position.
subplots_adjust(self[, left, right, bottom, ...])	Tune the subplot layout.
tight_layout(self[, pad, h_pad, w_pad, rect])	Adjust space between plot and figure.
timed([msg])	Timer decorator.
title(self, label, **kwargs)	Set the global title for current figure.

close(*self*, *fig=None*)
Close current or input figure.

Parameters

fig [CFigure or None] Handle to figure to close. If None (default), current figure is closed.

get_default_params(*self*)
Return current defaults for the figure.

Returns

default_parameters [dict] Contain default parameters value set.

get_state(*self*)
Returns the object state dictionary.

Returns

dict Dictionary containing the state of the object.

load_state(*self*, *path*)
Sets the object state from file.

Parameters

path [str] The full path of the file from which to load the object state.

See also:

set_state Sets the object state using input dictionary.

property n_sp
Returns the number of created subplots.

save_state(*self*, *path*)
Store the object state to file.

Parameters

path [str] Path of the file where to store object state.

Returns

str The full path of the stored object.

See also:

get_state Returns the object state dictionary.

savefig(*self*, *fname*, *dpi=None*, *facecolor='w'*, *edgecolor='w'*, *orientation='portrait'*, *papertype=None*, *file_format=None*, *transparent=False*, *bbox_inches=None*, *bbox_extra_artists=None*, *pad_inches=0.1*, *frameon=None*)
Save figure to disk.

Parameters

fname [string] containing a path to a filename, or a Python file-like object. If file_format is None and fname is a string, the output file_format is deduced from the extension of the filename. If the filename has no extension, the value of the rc parameter savefig.file_format is used. If fname is not a string, remember to specify file_format to ensure that the correct backend is used.

dpi [[None | scalar > 0], optional] The resolution in dots per inch. If None it will default to the value savefig.dpi in the matplotlibrc file.

facecolor, edgecolor [color or str, optional] The colors of the figure rectangle. Default 'w' (white).

orientation: ['landscape' | 'portrait'], optional not supported on all backends; currently only on postscript output

papertype [str, optional] One of 'letter', 'legal', 'executive', 'ledger', 'a0' through 'a10', 'b0' through 'b10'. Only supported for postscript output.

file_format [str, optional] One of the file extensions supported by the active backend. Most backends support png, pdf, ps, eps and svg.

transparent [bool, optional] If True, the axes patches will all be transparent; the figure patch will also be transparent unless facecolor and/or edgecolor are specified via kwargs. This is useful, for example, for displaying a plot on top of a colored background on a web page. The transparency of these patches will be restored to their original values upon exit of this function.

bbox_inches [scalar or str, optional] Bbox in inches. Only the given portion of the figure is saved. If 'tight', try to figure out the tight bbox of the figure.

bbox_extra_artists [list] A list of extra artists that will be considered when the tight bbox is calculated.

pad_inches [scalar] Amount of padding around the figure when bbox_inches is 'tight'.

frameon [bool, optional] If True, the figure patch will be colored, if False, the figure background will be transparent. If not provided, the rcParam 'savefig.frameon' will be used.

set_state(*self*, *state_dict*, *copy=False*)

Sets the object state using input dictionary.

Only readable attributes of the class, i.e. PUBLIC or READ/WRITE or READ ONLY, can be set.

If possible, a reference to the attribute to set is assigned. Use *copy=True* to always make a deepcopy before set.

Parameters

state_dict [dict] Dictionary containing the state of the object.

copy [bool, optional] By default (False) a reference to the attribute to assign is set. If True or a reference cannot be extracted, a deepcopy of the attribute is done first.

static show(*block=True*)

Show all created figures.

Parameters

block [boolean, default True] If true, execution is halted until the showed figure(s) are closed.

property sp

Return reference to active subplot class.

If no subplot is available, creates a new standard subplot in *(1,1,1)* position and returns its reference.

subplot(*self*, *n_rows=1*, *n_cols=1*, *grid_slot=1*, ***kwargs*)

Create a new subplot into specific position.

Creates a new subplot placing it in the n_plot position of the n_rows * n_cols imaginary grid. Position is indexed in raster-scan.

If subplot is created in a slot occupied by another subplot, old subplot will be used.

Parameters

n_rows [int] Number of rows of the imaginary grid used for subdividing the figure. Default 1 (one row).

n_cols [int] Number of columns of the imaginary grid used for subdividing the figure. Default 1 (one column).

grid_slot [int or tuple] If int, raster scan position of the new subplot. Default 1. If tuple, index of the slot of the grid. Each dimension can be specified as an int or a slice, e.g. in a 3 x 3 subplot grid, grid_slot=(0, slice(1, 3)) will create a subplot at row index 0 that spans between columns index 1 and 2.

Examples

```
import numpy as np
import matplotlib.pyplot as plt
from secml.figure import CFigure

fig = CFigure(fontsize=16)

# create a new subplot
fig.subplot(2, 2, 1)
x = np.linspace(-np.pi, np.pi, 100)
y = 2*np.sin(x)
# function `plot` will be applied to the last subplot created
fig.sp.plot(x, y)

# subplot indices are are the same of the first subplot
# so the following function will be run inside the previous plot
fig.subplot(2, 2, 1)
y = x
fig.sp.plot(x, y)

# create a new subplot
fig.subplot(2, 2, 3)
fig.sp.plot(x, y)

fig.subplot(2, 2, grid_slot=(1, slice(2)))
y = 2*np.sin(x)
fig.sp.plot(x, y)

plt.show()
```

subplots_adjust(*self*, *left=0.125*, *right=0.9*, *bottom=0.1*, *top=0.9*, *wspace=0.2*, *hspace=0.2*)

Tune the subplot layout.

Parameters

left [float, default 0.125] Left side of the subplots.

right [float, default 0.9] Right side of the subplots.

bottom [float, default 0.1] Bottom of the subplots.

top [float, default 0.9] Top of the subplots.

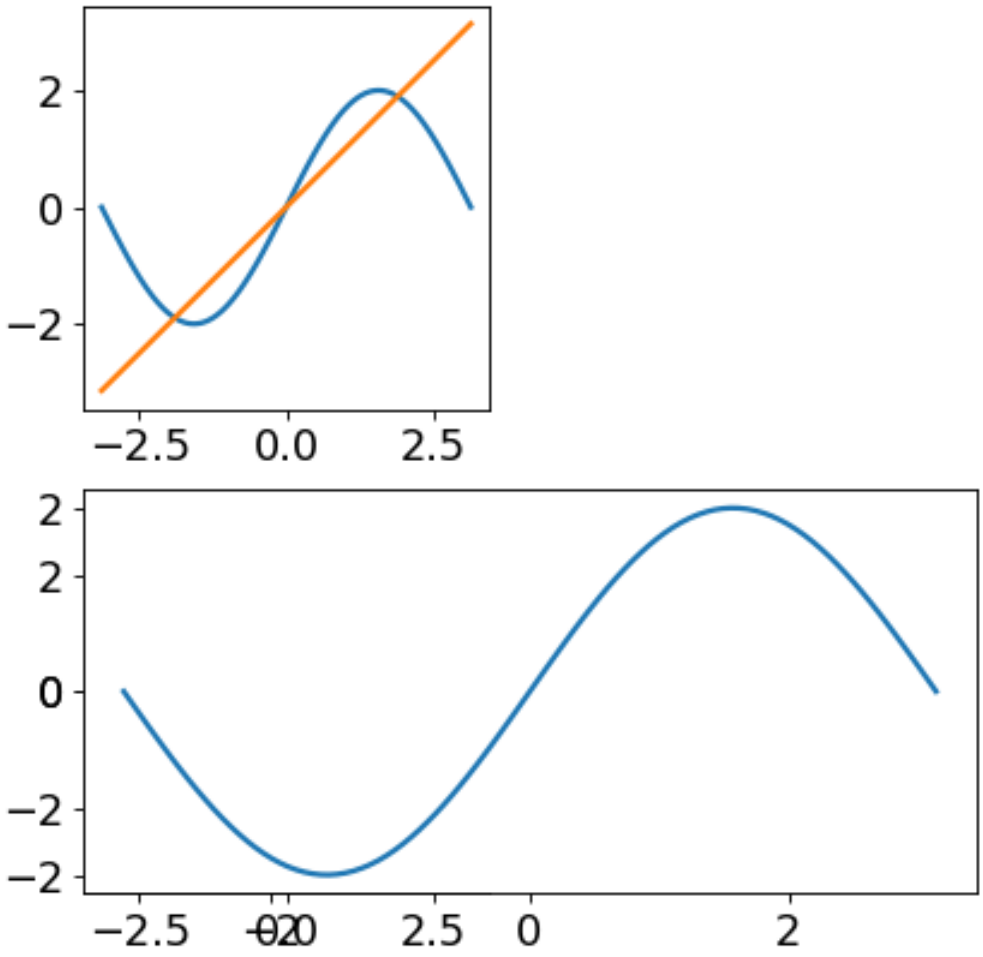
wspace [float, default 0.2] Amount of width reserved for blank space between subplots.

hspace [float, default 0.2] Amount of height reserved for white space between subplots

Examples

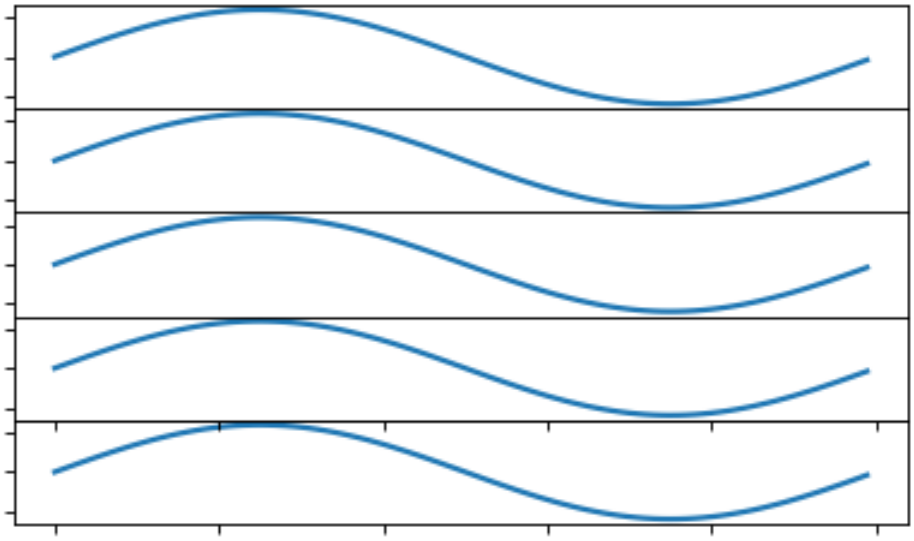
```
from secml.array import CArray
from secml.figure import CFigure

n = 5
fig = CFigure()

x = CArray.arange(100)
y = 3. * CArray.sin(x * 2. * 3.14 / 100.)
```

(continues on next page)


2
0
−2
−2.5
0.0
2.5
2
2
0
−2
−2
−2.5
2.5
0
2

(continued from previous page)

```
for i in range(n):
    temp = 510 + i
    sp = fig.subplot(n, 1, i)
    fig.sp.plot(x, y)
    # for add space from the figure's border you must increased default value␣
↪parameters
    fig.subplots_adjust(bottom=0.4, top=0.85, hspace=0.001)
    fig.sp.xticklabels(())
    fig.sp.yticklabels(())

fig.show()
```



tight_layout (*self*, *pad=1.08*, *h_pad=None*, *w_pad=None*, *rect=None*)
Adjust space between plot and figure.

Parameters

pad [float, default 1.08] Padding between the figure edge and the edges of subplots, as a fraction of the font-size.

h_pad, w_pad [float, defaults to pad_inches.] padding (height/width) between edges of adjacent subplots.

rect [tuple of scalars, default is (0, 0, 1, 1).] (left, bottom, right, top) in the normalized figure coordinate that the whole subplots area (including labels) will fit into.

title(*self*, *label*, ***kwargs*)

Set the global title for current figure.

Parameters

label [str] Text to use for the title.

****kwargs** Same as *text* method.

CPlot

All methods provided by *CPlot* and subclasses are available by calling the active subplot via *CFigure.sp*

Warning: To be never explicitly instanced. Will be created by *CFigure*.

class secml.figure._plots.c_plot.**CPlot**(*sp*, *default_params*)

Interface for standard plots.

This class provides an interface and few other methods useful for standard plot creation.

To be never explicitly instanced. Will be created by *CFigure*.

Parameters

sp [Axes] Subplot to use for plotting. Instance of *matplotlib.axes.Axes*.

default_params [dict] Dictionary with default parameters.

See also:

CFigure creates and handle figures.

Attributes

class_type Defines class type.

logger Logger for current object.

n_lines Returns the number of lines inside current subplot.

verbose Verbosity level of logger output.

Methods

bar(self, left, height[, width, bottom])	Bar plot.
barh(self, bottom, width[, height, left])	Horizontal bar plot.
boxplot(self, x[, notch, sym, vert, whis, ...])	Make a box and whisker plot.
clabel(self, contour, *args, **kwargs)	Label a contour plot.
colorbar(self, mappable[, ticks])	Add colorbar to plot.
contour(self, x, y, z, *args, **kwargs)	Draw contour lines of a function.
contourf(self, x, y, z, *args, **kwargs)	Draw filled contour of a function.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
errorbar(self, x, y[, xerr, yerr])	Plot with error deltas in yerr and xerr.
fill_between(self, x, y1[, y2, where, ...])	Fill the area between two horizontal curves.
get_class_from_type(class_type)	Return the class associated with input type.
get_legend(self)	Returns the handler of current subplot legend.
get_legend_handles_labels(self)	Return handles and labels for legend contained by the subplot.
get_lines(self)	Return a list of lines contained by the subplot.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
get_xticks_idx(self, xticks)	Returns the position of markers to plot.
grid(self[, grid_on, axis])	Draw grid for current plot.
hist(self, x, *args, **kwargs)	Plot a histogram.
imshow(self, img, *args, **kwargs)	Plot image.
legend(self, *args, **kwargs)	Create legend for plot.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
loglog(self, x[, y])	Plot with log scaling on both the x and y axis.
matshow(self, array, *args, **kwargs)	Plot an array as a matrix.
merge(self, sp)	Merge input subplot to active subplot.
plot(self, x[, y])	Plot a line.
plot_path(self, path[, path_style, ...])	Plot a path traversed by a point.
quiver(self, U, V[, X, Y, color, linestyle, ...])	A quiver plot displays velocity vectors as arrows with components (u,v) at the points (x,y).
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
scatter(self, x, y[, s, c])	Scatter plot of x vs y.
semilogx(self, x[, y])	Plot with log scaling on the x axis.
semilogy(self, x[, y])	Plot with log scaling on the y axis.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_axisbelow(self[, axisbelow])	Set axis ticks and gridlines below most artists.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
text(self, *args, **kwargs)	Create a Text instance at x, y with string text.
tick_params(self, *args, **kwargs)	Change the appearance of ticks and tick labels.

Continued on next page

Table 155 – continued from previous page

timed([msg])	Timer decorator.
title(self, text, *args, **kwargs)	Set a title for subplot.
xlabel(self, label, *args, **kwargs)	Set a label for the x axis.
xlim(self[, bottom, top])	Set axes x limits.
xscale(self, scale_type[, nonposx, basex])	Set scale for x axis.
xticklabels(self, labels, *args, **kwargs)	Set the xtick labels.
xticks(self, location_array, *args, **kwargs)	Set the x-tick locations and labels.
ylabel(self, label, *args, **kwargs)	Set a label for the y axis
ylim(self[, bottom, top])	Set axes y limits.
yscale(self, scale_type[, nonposy, basey])	Set scale for y axis.
yticklabels(self, labels, *args, **kwargs)	Set the ytick labels.
yticks(self, location_array, *args, **kwargs)	Set the y-tick locations and labels.

bar(*self*, *left*, *height*, *width=0.8*, *bottom=None*, **args*, ***kwargs*)

Bar plot.

Parameters

left [sequence of scalars] x coordinates of the left sides of the bars.

height [sequence of scalars] height(s) of the bars.

width [scalar or array-like, optional, default: 0.8] width(s) of the bars.

bottom [scalar or array-like, optional, default: None] y coordinate(s) of the bars.

color [scalar or array-like, optional] Colors of the bar faces.

edgecolor [scalar or array-like, optional] Colors of the bar edges.

linewidth [scalar or array-like, optional, default: None] Width of bar edge(s). If None, use default linewidth; If 0, don't draw edges.

xerr [scalar or array-like, optional, default: None] If not None, will be used to generate errorbar(s) on the bar chart.

yerr [scalar or array-like, optional, default: None] If not None, will be used to generate errorbar(s) on the bar chart.

ecolor [scalar or array-like, optional, default: None] Specifies the color of errorbar(s)

capsize [integer, optional, default: 3] Determines the length in points of the error bar caps.

error_kw [dict] dictionary of kwargs to be passed to errorbar method. ecolor and capsize may be specified here rather than independent kwargs.

align [['edge' | 'center'], optional, default: 'edge'] If edge, aligns bars by their left edges (for vertical bars) and by their bottom edges (for horizontal bars). If center, interpret the left argument as the coordinates of the centers of the bars.

orientation ['vertical' | 'horizontal', optional, default: 'vertical'] The orientation of the bars.

log [boolean, optional, default: False] If true, sets the axis to be log scale.

Returns

bar_list [list of bar type objects]

Examples

```
from secml.array import CArray
from secml.figure import CFigure

fig = CFigure(fontsize=12)

n = 12
X = CArray.arange(n)
Y1 = (1 - X / float(n)) * (1.0 - 0.5) * CArray.rand((n,)) + 0.5
Y2 = (1 - X / float(n)) * (1.0 - 0.5) * CArray.rand((n,)) + 0.5

fig.sp.xticks([0.025, 0.025, 0.95, 0.95])
fig.sp.bar(X, Y1, facecolor='#9999ff', edgecolor='white')
fig.sp.bar(X, -Y2, facecolor='#ff9999', edgecolor='white')

for x, y in zip(X, Y1):
    fig.sp.text(x, y, '%.2f' % y, ha='center', va='bottom')

for x, y in zip(X, Y2):
    fig.sp.text(x, -y - 0.02, '%.2f' % y, ha='center', va='top')

fig.sp.xlim(-.5, n-.5)
fig.sp.xticks(())
fig.sp.ylim(-1.25, 1.25)
fig.sp.yticks(())

fig.sp.grid()
fig.show()
```

barh (*self*, *bottom*, *width*, *height=0.8*, *left=None*, **args*, ***kwargs*)

Horizontal bar plot.

Parameters

bottom [sequence of scalars] y coordinates of the bars.

width [sequence of scalars] width(s) of the bars.

height [scalar or array-like, optional, default: 0.8] height(s) of the bars.

left [scalar or array-like, optional, default: None] x coordinate(s) of the bars.

color [scalar or array-like, optional] Colors of the bar faces.

edgecolor [scalar or array-like, optional] Colors of the bar edges.

linewidth [scalar or array-like, optional, default: None] Width of bar edge(s). If None, use default linewidth; If 0, don't draw edges.

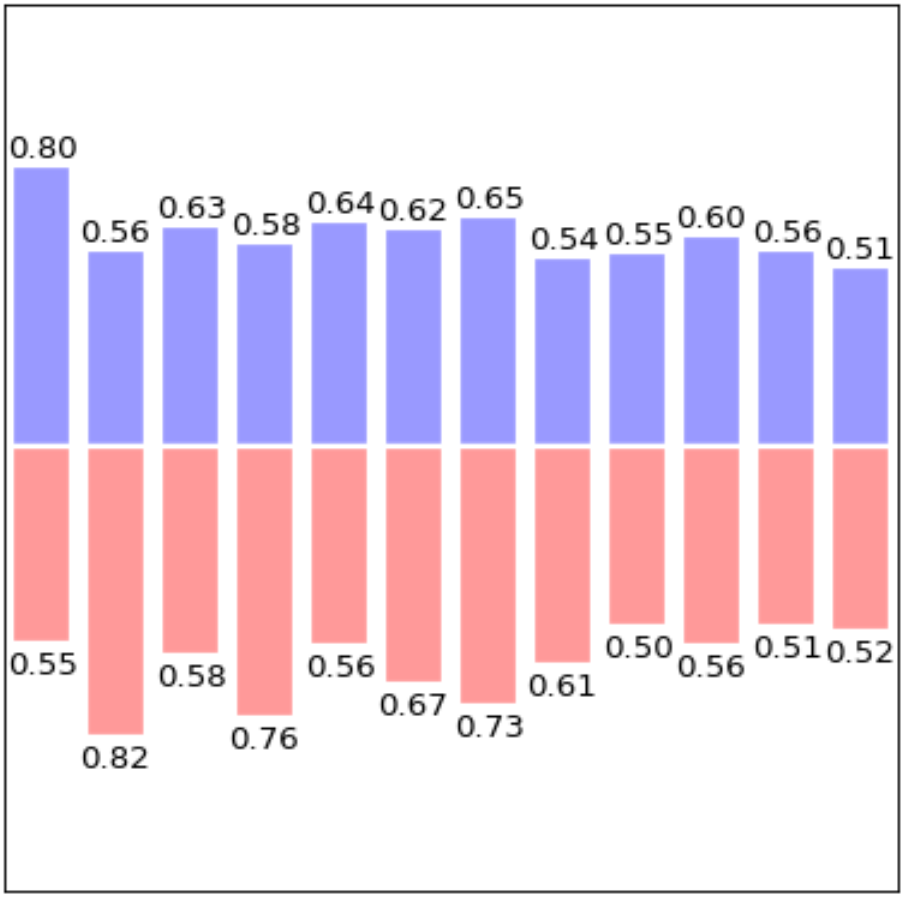
xerr [scalar or array-like, optional, default: None] If not None, will be used to generate errorbar(s) on the bar chart.

yerr [scalar or array-like, optional, default: None] If not None, will be used to generate errorbar(s) on the bar chart.

ecolor [scalar or array-like, optional, default: None] Specifies the color of errorbar(s)

capsize [integer, optional, default: 3] Determines the length in points of the error bar caps.

error_kw [dict] dictionary of kwargs to be passed to errorbar method. ecolor and capsize may be specified here rather than independent kwargs.


0.80
0.56
0.63
0.58
0.64
0.62
0.65
0.54
0.55
0.60
0.56
0.51
0.55
0.82
0.58
0.76
0.56
0.67
0.73
0.61
0.50
0.56
0.51
0.52

align [['edge' | 'center'], optional, default: 'edge'] If edge, aligns bars by their left edges (for vertical bars) and by their bottom edges (for horizontal bars). If center, interpret the left argument as the coordinates of the centers of the bars.

orientation ['vertical' | 'horizontal', optional, default: 'vertical'] The orientation of the bars.

log [boolean, optional, default: False] If true, sets the axis to be log scale.

Returns

bar_list [list of bar type objects]

boxplot(*self*, *x*, *notch=False*, *sym=None*, *vert=True*, *whis=1.5*, *positions=None*, *widths=None*, *patch_artist=False*, *bootstrap=None*, *usermedians=None*, *conf_intervals=None*, *meanline=False*, *showmeans=False*, *showcaps=True*, *showbox=True*, *showfliers=True*, *boxprops=None*, *labels=None*, *flierprops=None*, *medianprops=None*, *meanprops=None*, *capprops=None*, *whiskerprops=None*, *manage_xticks=True*)

Make a box and whisker plot.

Make a box and whisker plot for each column of *x* or each vector in sequence *x*. The box extends from the lower to upper quartile values of the data, with a line at the median. The whiskers extend from the box to show the range of the data. Flier points are those past the end of the whiskers.

Parameters

x [Array or a sequence of vectors.] The input data.

notch [bool, default = False] If False, produces a rectangular box plot. If True, will produce a notched box plot

sym [str or None, default = None] The default symbol for flier points. Enter an empty string ('') if you don't want to show fliers. If *None*, then the fliers default to 'b+' If you want more control use the flierprops kwarg.

vert [bool, default = True] If True (default), makes the boxes vertical. If False, makes horizontal boxes.

whis [float, sequence (default = 1.5) or string] As a float, determines the reach of the whiskers past the first and third quartiles (e.g., Q3 + whis*IQR, IQR = interquartile range, Q3-Q1). Beyond the whiskers, data are considered outliers and are plotted as individual points. Set this to an unreasonably high value to force the whiskers to show the min and max values. Alternatively, set this to an ascending sequence of percentile (e.g., [5, 95]) to set the whiskers at specific percentiles of the data. Finally, *whis* can be the string 'range' to force the whiskers to the min and max of the data. In the edge case that the 25th and 75th percentiles are equivalent, *whis* will be automatically set to 'range'.

bootstrap [None (default) or integer] Specifies whether to bootstrap the confidence intervals around the median for notched boxplots. If bootstrap==None, no bootstrapping is performed, and notches are calculated using a Gaussian-based asymptotic approximation (see McGill, R., Tukey, J.W., and Larsen, W.A., 1978, and Kendall and Stuart, 1967). Otherwise, bootstrap specifies the number of times to bootstrap the median to determine it's 95% confidence intervals. Values between 1000 and 10000 are recommended.

usermedians [array-like or None (default)] An array or sequence whose first dimension (or length) is compatible with *x*. This overrides the medians computed by matplotlib for each element of *usermedians* that is not None. When an element of *usermedians* == None, the median will be computed by matplotlib as normal.

conf_intervals [array-like or None (default)] Array or sequence whose first dimension (or length) is compatible with *x* and whose second dimension is 2. When the current element of *conf_intervals* is not None, the notch locations computed by matplotlib are overridden

(assuming notch is True). When an element of *conf_intervals* is None, boxplot compute notches the method specified by the other kwargs (e.g., *bootstrap*).

positions [array-like, default = [1, 2, ..., n]] Sets the positions of the boxes. The ticks and limits are automatically set to match the positions.

widths [array-like, default = 0.5] Either a scalar or a vector and sets the width of each box. The default is 0.5, or `0.15*(distance between extreme positions)` if that is smaller.

labels [sequence or None (default)] Labels for each dataset. Length must be compatible with dimensions of *x*

patch_artist [bool, default = False] If False produces boxes with the Line2D artist If True produces boxes with the Patch artist

showmeans [bool, default = False] If True, will toggle one the rendering of the means

showcaps [bool, default = True] If True, will toggle one the rendering of the caps

showbox [bool, default = True] If True, will toggle one the rendering of box

showfliers [bool, default = True] If True, will toggle one the rendering of the fliers

boxprops [dict or None (default)] If provided, will set the plotting style of the boxes

whiskerprops [dict or None (default)] If provided, will set the plotting style of the whiskers

capprops [dict or None (default)] If provided, will set the plotting style of the caps

flierprops [dict or None (default)] If provided, will set the plotting style of the fliers

medianprops [dict or None (default)] If provided, will set the plotting style of the medians

meanprops [dict or None (default)] If provided, will set the plotting style of the means

meanline [bool, default = False] If True (and *showmeans* is True), will try to render the mean as a line spanning the full width of the box according to *meanprops*. Not recommended if *shownotches* is also True. Otherwise, means will be shown as points.

Returns

result [dict] A dictionary mapping each component of the boxplot to a list of the `matplotlib.lines.Line2D` instances created. That dictionary has the following keys (assuming vertical boxplots):

- boxes: the main body of the boxplot showing the quartiles and the median's confidence intervals if enabled.
- medians: horizonal lines at the median of each box.
- whiskers: the vertical lines extending to the most extreme, n-outlier data points.
- caps: the horizontal lines at the ends of the whiskers.
- fliers: points representing data that extend beyond the whiskers (outliers).
- means: points or lines representing the means.

clabel (*self*, *contour*, **args*, ***kwargs*)

Label a contour plot.

Parameters

contour [contour object] returned from contour function

fontsize [int] size in points or relative size e.g., 'smaller', 'x-large'

colors [str] if None, the color of each label matches the color of the corresponding contour if one string color, e.g., colors = 'r' or colors = 'red', all labels will be plotted in this color if a tuple of matplotlib color args (string, float, rgb, etc), different labels will be plotted in different colors in the order specified

inline [bool] controls whether the underlying contour is removed or not. Default is True.

inline_spacing [int] space in pixels to leave on each side of label when placing inline. Defaults to 5. This spacing will be exact for labels at locations where the contour is straight, less so for labels on curved contours.

fmt [str] a format string for the label. Default is '%1.3f' Alternatively, this can be a dictionary matching contour levels with arbitrary strings to use for each contour level (i.e., fmt[level]=string), or it can be any callable, such as a Formatter instance, that returns a string when called with a numeric contour level.

manual [bool] if True, contour labels will be placed manually using mouse clicks. Click the first button near a contour to add a label, click the second button (or potentially both mouse buttons at once) to finish adding labels. The third button can be used to remove the last label added, but only if labels are not inline. Alternatively, the keyboard can be used to select label locations (enter to end label placement, delete or backspace act like the third mouse button, and any other key will select a label location). manual can be an iterable object of x,y tuples. Contour labels will be created as if mouse is clicked at each x,y positions.

rightside_up [bool] if True (default), label rotations will always be plus or minus 90 degrees from level.

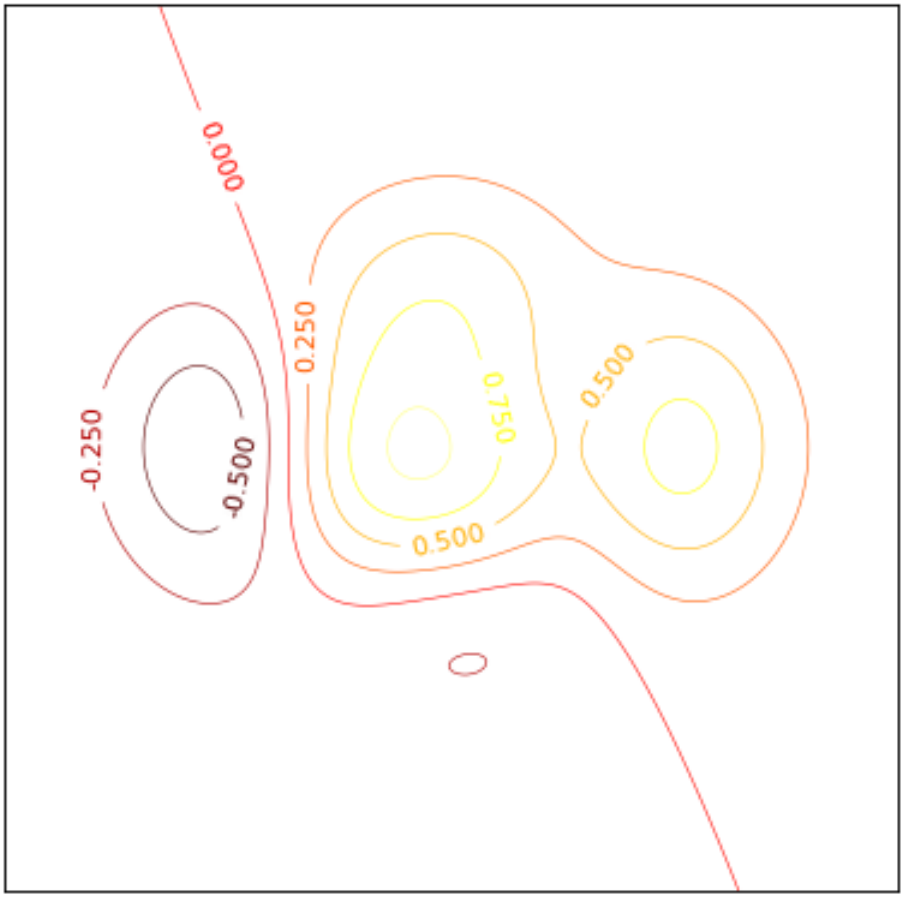
Examples

```
from secml.array import CArray
from secml.figure import CFigure


def f(x, y):
    return (1 - x / 2 + x ** 5 + y ** 3) * (-x ** 2 - y ** 2).exp()


fig = CFigure()

x_linspace = CArray.linspace(-3, 3, 256)
y_linspace = CArray.linspace(-3, 3, 256)

X, Y = CArray.meshgrid((x_linspace, y_linspace))

C = fig.sp.contour(X, Y, f(X, Y), linewidths=.5, cmap='hot')
fig.sp.clabel(C, inline=1, fontsize=10)

fig.sp.xticks(())
fig.sp.yticks(())

fig.show()
```

colorbar(*self*, *mappable*, *ticks=None*, **args*, ***kwargs*)

Add colorbar to plot.

Parameters


0.000
0.250
0.500
0.750
0.500
-0.250
-0.500
0.500
0

mappable [object] Image, ContourSet, or other to which the colorbar applies

use_gridspec [boolean, default False] if True colorbar is created as an instance of Subplot using the grid_spec module.

Additional keyword arguments are of two kinds:

Axes properties:

Property	Description
orientation	vertical or horizontal
fraction, default 0.15	fraction of original axes to use for colorbar
pad, default 0.05 if vertical, 0.15 if horizontal	fraction of original axes between colorbar and new image axes
shrink, default 1.0	fraction by which to shrink the colorbar
aspect, default 20	ratio of long to short dimensions
anchor, default (0.0, 0.5) if vertical; (0.5, 1.0) if horizontal	the anchor point of the colorbar axes
panchor, default (1.0, 0.5) if vertical; (0.5, 0.0) if horizontal;	the anchor point of the colorbar parent axes. If False, the parent axes' anchor will be unchanged

Colorbar properties:

Property	Description
extend	['neither' | 'both' | 'min' | 'max'] If not 'neither', make pointed end(s) for out-of- range values. These are set for a given colormap using the colormap set_under and set_over methods.
extendfrac	[None | 'auto' | length | lengths] If set to None, both the minimum and maximum triangular colorbar extensions with have a length of 5% of the interior colorbar length (this is the default setting). If set to 'auto', makes the triangular colorbar extensions the same lengths as the interior boxes (when spacing is set to 'uniform') or the same lengths as the respective adjacent interior boxes (when spacing is set to 'proportional'). If a scalar, indicates the length of both the minimum and maximum triangular colorbar extensions as a fraction of the interior colorbar length. A two-element sequence of fractions may also be given, indicating the lengths of the minimum and maximum colorbar extensions respectively as a fraction of the interior colorbar length.
extendrect	[False | True] If False the minimum and maximum colorbar extensions will be triangular (the default). If True the extensions will be rectangular.
spacing	['uniform' | 'proportional'] Uniform spacing gives each discrete color the same space; proportional makes the space proportional to the data interval.
ticks	[None | list of ticks | Locator object] If None, ticks are determined automatically from the input.
format	[None | format string | Formatter object] If None, the ScalarFormatter is used. If a format string is given, e.g., '%.3f', that is used. An alternative Formatter object may be given instead.
drawedges	[False | True] If true, draw lines at color boundaries.

Notes

If mappable is a ContourSet, its extend kwarg is included automatically. Note that the shrink kwarg provides a simple way to keep a vertical colorbar. If the colorbar is too tall (or a horizontal colorbar is too wide) use a smaller value of shrink.

Examples

```
from secml.array import CArray
from secml.figure import CFigure


def f(x, y):
    return (1 - x / 2 + x ** 5 + y ** 3) * (-x ** 2 - y ** 2).exp()

fig = CFigure(width=10, title="Colorbar Example")
fig.subplot(1, 2, 1)

x_linspace = CArray.linspace(-3, 3, 256)
y_linspace = CArray.linspace(-3, 3, 256)

X, Y = CArray.meshgrid((x_linspace, y_linspace))
c = fig.sp.contourf(X, Y, f(X, Y), 8, alpha=.75, cmap='hot')
fig.sp.colorbar(c)
fig.sp.title("Hot Contourf")
fig.sp.xticks(())
fig.sp.yticks(())

fig.subplot(1, 2, 2)
c = fig.sp.contourf(X, Y, f(X, Y), 8, alpha=.75, cmap='winter')
fig.sp.colorbar(c)
fig.sp.title("Cold Contourf")
fig.sp.xticks(())
fig.sp.yticks(())

fig.show()
```

contour (*self*, *x*, *y*, *z*, **args*, ***kwargs*)
Draw contour lines of a function.

Parameters

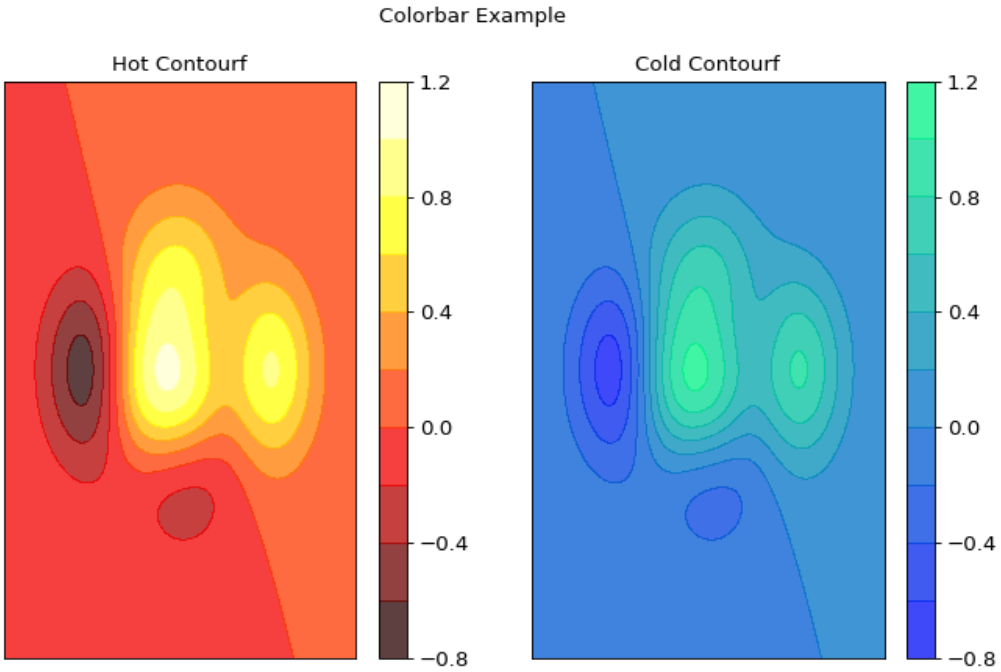
x, y [CArray or list] specify the (x, y) coordinates of the surface. X and Y must both be 2-D with the same shape as Z, or they must both be 1-D such that len(X) is the number of columns in Z and len(Y) is the number of rows in Z.

z [CArray or list] value into (x, y) surface's position

colors [[None | string | (mpl_colors)]] If None, the colormap specified by cmap will be used. If a string, like 'r' or 'red', all levels will be plotted in this color. If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

alpha [float] The alpha blending value

cmap [[None | Colormap]] A cm Colormap instance or None. If cmap is None and colors is None, a default Colormap is used.



vmin, vmax [[None | scalar]] If not None, either or both of these values will be supplied to the matplotlib.colors. Normalize instance, overriding the default color scaling based on levels.

levels [[level0, level1, …, leveln]] A list of floating point numbers indicating the level curves to draw; e.g., to draw just the zero contour pass levels=[0]

origin [[None | 'upper' | 'lower' | 'image']] If None, the first value of Z will correspond to the lower left corner, location (0,0). If 'image', the default parameter value for image.origin will be used. This keyword is not active if X and Y are specified in the call to contour.

extent [[None | (x0,x1,y0,y1)]] If origin is not None, then extent is interpreted as in matplotlib.pyplot.imshow(): it gives the outer pixel boundaries. In this case, the position of Z[0,0] is the center of the pixel, not a corner. If origin is None, then (x0, y0) is the position of Z[0,0], and (x1, y1) is the position of Z[-1,-1]. This keyword is not active if X and Y are specified in the call to contour.

extend [['neither' | 'both' | 'min' | 'max']] Unless this is 'neither', contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range.

antialiased [[True | False]] enable antialiasing, overriding the defaults. For filled contours, the default is True. For line contours, it is taken from default_parameters ['lines.antialiased'].

linewidths [[None | number | tuple of numbers]] If linewidths is None, the default width in lines.linewidth default_parameters is used. If a number, all levels will be plotted with this linewidth. If a tuple, different levels will be plotted with different linewidths in the order specified.

linestyles [[None | 'solid' | 'dashed' | 'dashdot' | 'dotted']] If linestyles is None, the default is 'solid' unless the lines are monochrome. In that case, negative contours will take their linestyle from the matplotlibrc *contour.negative_* linestyle setting. linestyles can also be an iterable of the above strings specifying a set of linestyles to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

Examples

```
from secml.array import CArray
from secml.figure import CFigure


def f(x, y):
    return (1 - x / 2 + x ** 5 + y ** 3) * (-x ** 2 - y ** 2).exp()

fig = CFigure()

x_linspace = CArray.linspace(-3, 3, 256)
y_linspace = CArray.linspace(-3, 3, 256)

X, Y = CArray.meshgrid((x_linspace, y_linspace))

C = fig.sp.contour(X, Y, f(X, Y), linewidths=.5, cmap='hot')

fig.sp.xticks(())
fig.sp.yticks(())

fig.show()
```

contourf (*self*, *x*, *y*, *z*, **args*, ***kwargs*)
Draw filled contour of a function.

Parameters

x, y [CArray or list] specify the (x, y) coordinates of the surface. X and Y must both be 2-D with the same shape as Z, or they must both be 1-D such that len(X) is the number of columns in Z and len(Y) is the number of rows in Z.

z [CArray or list] value into (x, y) surface's position

colors [[None | string | (mpl_colors)]] If None, the colormap specified by cmap will be used. If a string, like 'r' or 'red', all levels will be plotted in this color. If a tuple of matplotlib color args (string, float, rgb, etc), different levels will be plotted in different colors in the order specified.

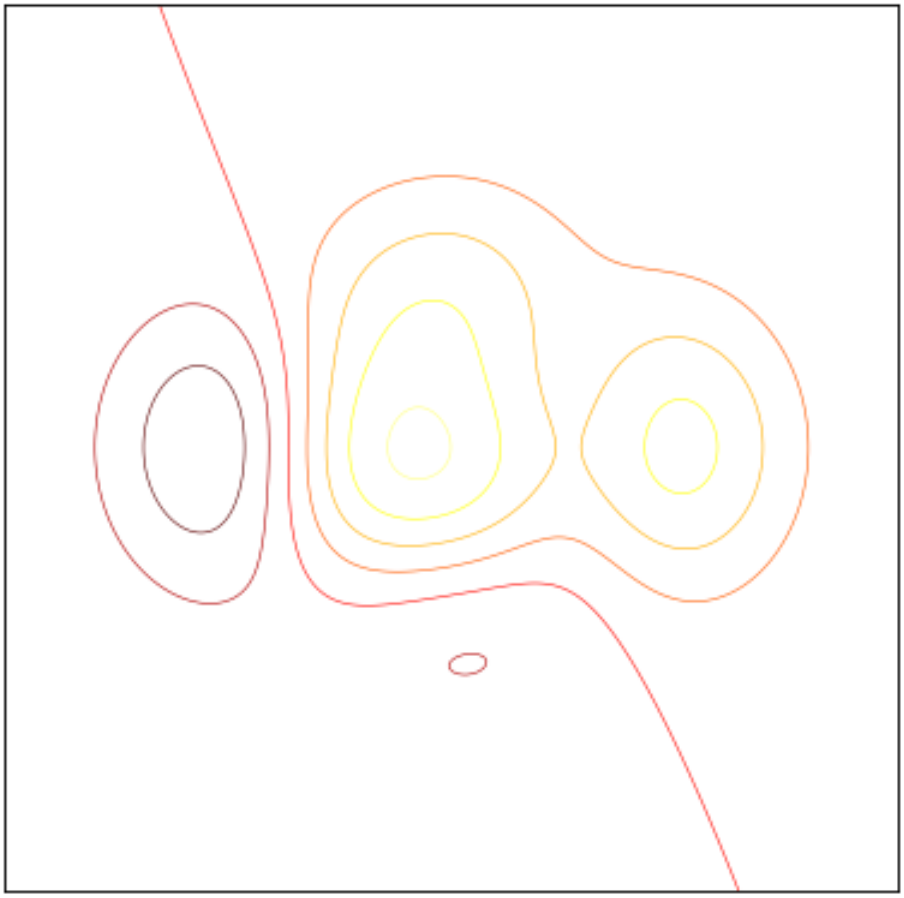
alpha [float] The alpha blending value

cmap [[None | Colormap]] A cm Colormap instance or None. If cmap is None and colors is None, a default Colormap is used.

vmin, vmax [[None | scalar]] If not None, either or both of these values will be supplied to the matplotlib.colors. Normalize instance, overriding the default color scaling based on levels.

levels [[level0, level1, ..., leveln]] A list of floating point numbers indicating the level curves to draw; e.g., to draw just the zero contour pass levels=[0]

origin [[None | 'upper' | 'lower' | 'image']] If None, the first value of Z will correspond to the lower left corner, location (0,0). If 'image', the default parameter value for im-



age.origin will be used. This keyword is not active if X and Y are specified in the call to contour.

extent [[None | (x0,x1,y0,y1)]] If origin is not None, then extent is interpreted as in matplotlib.pyplot.imshow(): it gives the outer pixel boundaries. In this case, the position of Z[0,0] is the center of the pixel, not a corner. If origin is None, then (x0, y0) is the position of Z[0,0], and (x1, y1) is the position of Z[-1,-1]. This keyword is not active if X and Y are specified in the call to contour.

extend [['neither' | 'both' | 'min' | 'max']] Unless this is 'neither', contour levels are automatically added to one or both ends of the range so that all data are included. These added ranges are then mapped to the special colormap values which default to the ends of the colormap range.

antialiased [[True | False]] enable antialiasing, overriding the defaults. For filled contours, the default is True. For line contours, it is taken from default_parameters ['lines.antialiased'].

Examples

```
from secml.array import CArray
from secml.figure import CFigure


def f(x, y):
    return (1 - x / 2 + x ** 5 + y ** 3) * (-x ** 2 - y ** 2).exp()

fig = CFigure()

x_linspace = CArray.linspace(-3, 3, 256)
y_linspace = CArray.linspace(-3, 3, 256)

X, Y = CArray.meshgrid((x_linspace, y_linspace))
fig.sp.contourf(X, Y, f(X, Y), 8, alpha=.75, cmap='hot')

fig.sp.xticks(())
fig.sp.yticks(())

fig.show()
```

errorbar(*self*, *x*, *y*, *xerr=None*, *yerr=None*, **args*, ***kwargs*)

Plot with error deltas in yerr and xerr.

Vertical errorbars are plotted if yerr is not None. Horizontal errorbars are plotted if xerr is not None. x, y, xerr, and yerr can all be scalars, which plots a single error bar at x, y.

Parameters

x [list or CArray] x axis values.

y [list or CArray] y axis values.

xerr, yerr [[scalar | N, Nx1, or 2xN array-like], default None] If a scalar number, len(N) array-like object, or an Nx1 array-like object, errorbars are drawn at +/-value relative to the data. If a sequence of shape 2xN, errorbars are drawn at -row1 and +row2 relative to the data.

fmt [['' | 'none' | plot format string], default ''] The plot format symbol. If fmt is 'none' (case-insensitive), only the errorbars are plotted. This is used for adding errorbars to a bar

plot, for example. Default is ‘’, an empty plot format string; properties are then identical to the defaults for plot().

ecolor [[None | mpl color], default None] A matplotlib color arg which gives the color the errorbar lines; if None, use the color of the line connecting the markers.

elinewidth [scalar, default None] The linewidth of the errorbar lines. If None, use the linewidth.

capsize [scalar, default 3] The length of the error bar caps in points.

capthick [scalar, default None] An alias kwarg to markeredgewidth (a.k.a. - mew). This setting is a more sensible name for the property that controls the thickness of the error bar cap in points. For backwards compatibility, if mew or markeredgewidth are given, then they will over-ride capthick. This may change in future releases.

barsabove [[True | False]] if True, will plot the errorbars above the plot symbols. Default is below.

lolims, uplims, xlolims, xuplims [[False | True], default False] These arguments can be used to indicate that a value gives only upper/lower limits. In that case a caret symbol is used to indicate this. lims-arguments may be of the same type as xerr and yerr. To use limits with inverted axes, set_xlim() or set_ylim() must be called before errorbar().

errorevery [positive integer, default 1] subsamples the errorbars. e.g., if everyerror=5, errorbars for every 5-th datapoint will be plotted. The data plot itself still shows all data points.

Examples

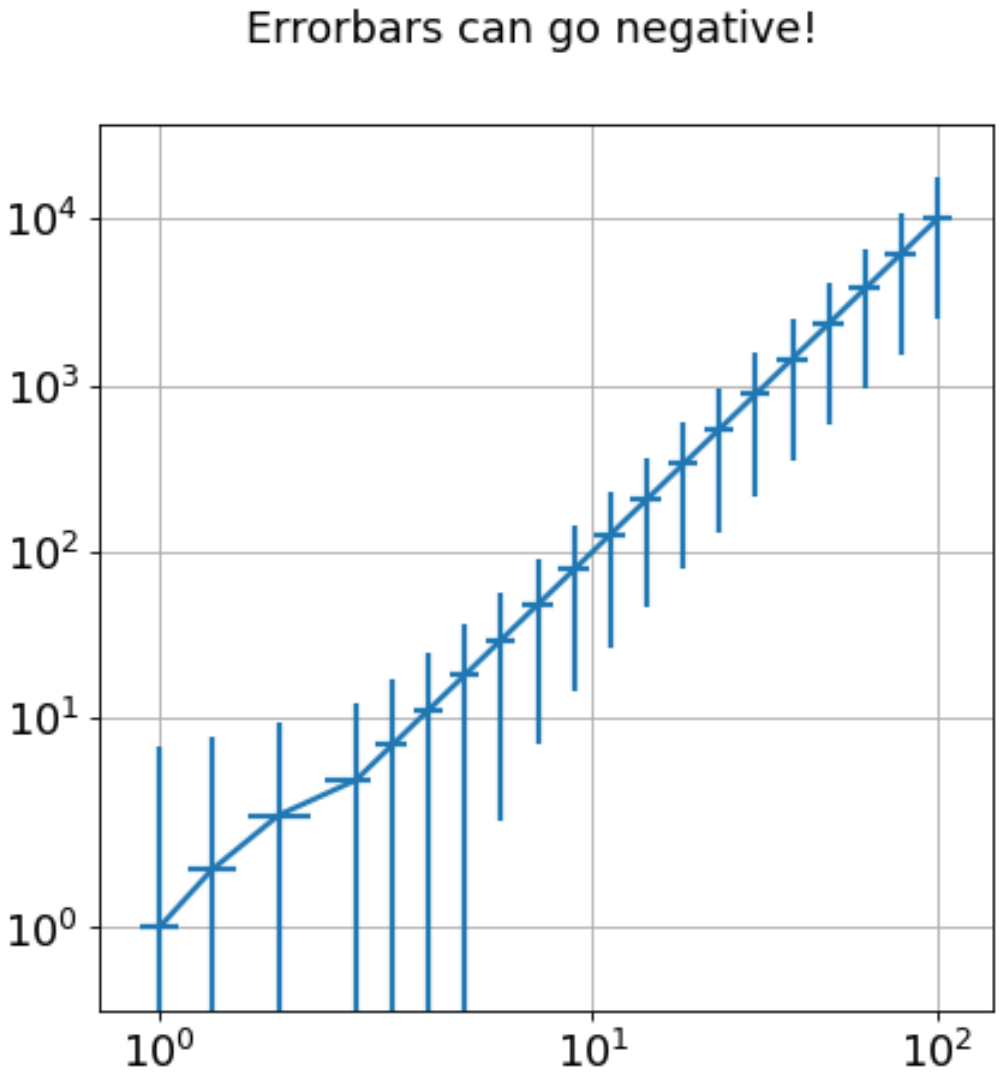
```
from secml.array import CArray
from secml.figure import CFigure

fig = CFigure(fontsize=16)
fig.title('Errorbars can go negative!')

fig.sp.xscale("symlog", nonposx='clip')
fig.sp.yscale("symlog", nonposy='clip')

x = CArray(10.0).pow(CArray.linspace(0.0, 2.0, 20))
y = x ** 2.0

fig.sp.errorbar(x, y, xerr=0.1 * x, yerr=5.0 + 0.75 * y)

fig.sp.ylim(bottom=0.1)

fig.sp.grid()
fig.show()
```

fill_between (*self*, *x*, *y1*, *y2=0*, *where=None*, *interpolate=False*, *step=None*, ***kwargs*)
Fill the area between two horizontal curves.

The curves are defined by the points (x, y1) and (x, y2). This creates one or multiple polygons describing the filled area.

You may exclude some horizontal sections from filling using where.

By default, the edges connect the given points directly. Use step if the filling should be a step function, i.e. constant in between x.


Errorbars can go negative!
10^4
10^3
10^2
10^1
10^0
10^0
10^1
10^2

Parameters

x [CArray (length N)] The x coordinates of the nodes defining the curves.

y1 [CArray (length N) or scalar] The y coordinates of the nodes defining the first curve.

y2 [CArray (length N) or scalar, optional, default: 0] The y coordinates of the nodes defining the second curve.

where [CArray of bool (length N), optional, default: None] Define where to exclude some horizontal regions from being filled. The filled regions are defined by the coordinates x[where]. More precisely, fill between x[i] and x[i+1] if where[i] and where[i+1]. Note that this definition implies that an isolated True value between two False values in where will not result in filling. Both sides of the True position remain unfilled due to the adjacent False values.

interpolate [bool, optional] This option is only relvant if where is used and the two curves are crossing each other. Semantically, where is often used for y1 > y2 or similar. By default, the nodes of the polygon defining the filled region will only be placed at the positions in the x array. Such a polygon cannot describe the above semantics close to the intersection. The x-sections containing the intersecion are simply clipped. Setting interpolate to True will calculate the actual intersection point and extend the filled region up to this point.

step [{'pre', 'post', 'mid'}, optional] Define step if the filling should be a step function, i.e. constant in between x. The value determines where the step will occur:

- 'pre': The y value is continued constantly to the left from every x position, i.e. the interval (x[i-1], x[i]] has the value y[i].
- 'post': The y value is continued constantly to the right from every x position, i.e. the interval [x[i], x[i+1]) has the value y[i].
- 'mid': Steps occur half-way between the x positions.

get_legend(*self*)
Returns the handler of current subplot legend.

get_legend_handles_labels(*self*)
Return handles and labels for legend contained by the subplot.

get_lines(*self*)
Return a list of lines contained by the subplot.

get_params(*self*)
Returns the dictionary of class hyperparameters.

A hyperparameter is a PUBLIC or READ/WRITE attribute.

get_state(*self*)
Returns the object state dictionary.

Returns

dict Dictionary containing the state of the object.

get_xticks_idx(*self*, *xticks*)
Returns the position of markers to plot.

Parameters

xticks [CArray] Ticks of x-axis where marker should be plotted.

Returns

ticks_idx [list] List with the position of each xtick.

Notes

If a given xtick is not exactly available, the closest value's position will be returned.

grid(*self*, *grid_on=True*, *axis='both'*, ***kwargs*)
Draw grid for current plot.

Parameters

grid_on [boolean, default True] if True show grid, elsewhere hide grid.

axis [string, default 'both'] can be 'both' (default), 'x', or 'y' to control which set of gridlines are drawn.

kwargs [any] Other keyword arguments for grid.

Examples

```
from secml.array import CArray
from secml.figure import CFigure

X = CArray.linspace(-3.14, 3.14, 256, endpoint=True)
C, S = X.cos(), X.sin()

fig = CFigure(fontsize=14)

fig.sp.plot(X, C, color='red', alpha=0.5, linewidth=1.0, linestyle='-', label=
↪"cosine")
fig.sp.plot(X, S, label="sine")

fig.sp.xticks(CArray([-3.14, -3.14 / 2, 0, 3.14 / 2, 3.14]))
fig.sp.yticks(CArray([-1, 0, +1]))
fig.sp.grid()
fig.sp.legend(loc=0)

fig.show()
```

hist(*self*, *x*, **args*, ***kwargs*)
Plot a histogram.

Compute and draw the histogram of x.

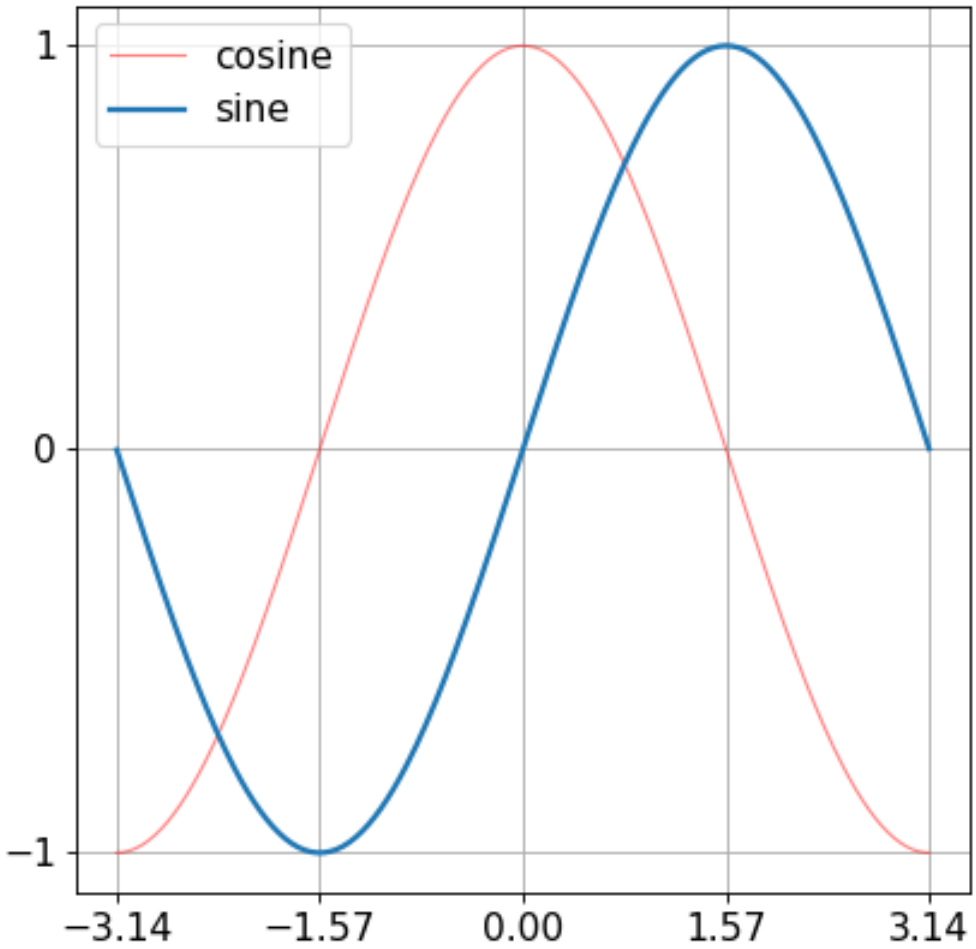
The return value is a tuple (n, bins, patches) or ([n0, n1, ...], bins, [patches0, patches1,...]) if the input contains multiple data.

Multiple data can be provided via x as a list of datasets of potentially different length ([x0, x1, ...]), or as a 2-D ndarray in which each column is a dataset.

Parameters

x [(n,) array or sequence of (n,) arrays] Input values, this takes either a single array or a sequency of arrays which are not required to be of the same length

bins [integer or array_like, optional, default is 10] If an integer is given, bins + 1 bin edges are returned. Unequally spaced bins are supported if bins is a sequence.


cosine
sine
1
0
−1
−3.14
−1.57
0.00
1.57
3.14

range [tuple or None, optional] The lower and upper range of the bins. Lower and upper outliers are ignored. If not provided, range is (x.min(), x.max()). Range has no effect if bins is a sequence. If bins is a sequence or range is specified, autoscaling is based on the specified bin range instead of the range of x.

density [boolean, optional] If True, the first element of the return tuple will be the counts normalized to form a probability density, i.e., n/(len(x)`dbin), i.e., the integral of the histogram will sum to 1. If stacked is also True, the sum of the histograms is normalized to 1.

weights [(n,) array_like or None, optional] An array of weights, of the same shape as x. Each value in x only contributes its associated weight towards the bin count (instead of 1). If density is True, the weights are normalized, so that the integral of the density over the range remains 1.

cumulative [boolean, optional] Dafault False. If True, then a histogram is computed where each bin gives the counts in that bin plus all bins for smaller values. The last bin gives the total number of datapoints. If density is also True then the histogram is normalized such that the last bin equals 1. If cumulative evaluates to less than 0 (e.g., -1), the direction of accumulation is reversed. In this case, if density is also True, then the histogram is normalized such that the first bin equals 1.

bottom [array_like, scalar, or None] Location of the bottom baseline of each bin. If a scalar, the base line for each bin is shifted by the same amount. If an array, each bin is shifted independently and the length of bottom must match the number of bins. If None, defaults to 0.

histtype [{'bar', 'barstacked', 'step', 'stepfilled'}, optional]

- 'bar' (default) is a traditional bar-type histogram. If multiple data are given the bars are aranged side by side.
- 'barstacked' is a bar-type histogram where multiple data are stacked on top of each other.
- 'step' generates a lineplot that is by default unfilled.
- 'stepfilled' generates a lineplot that is by default filled.

align [{'left', 'mid', 'right'}, optional]

- 'left': bars are centered on the left bin edges.
- 'mid': default, bars are centered between the bin edges.
- 'right': bars are centered on the right bin edges.

orientation [{'horizontal', 'vertical'}, optional] If 'horizontal', barh will be used for bar-type histograms and the bottom kwarg will be the left edges.

rwidth [scalar or None, optional] The relative width of the bars as a fraction of the bin width. If None, automatically compute the width. Ignored if histtype is 'step' or 'stepfilled'.

log [boolean, optional] Default False. If True, the histogram axis will be set to a log scale. If log is True and x is a 1D array, empty bins will be filtered out and only the non-empty (n, bins, patches) will be returned.

color [color or array_like of colors or None, optional] Color spec or sequence of color specs, one per dataset. Default (None) uses the standard line color sequence.

label [string or None, optional] String, or sequence of strings to match multiple datasets. Bar charts yield multiple patches per dataset, but only the first gets the label, so that the legend command will work as expected.

stacked [boolean, optional] If True, multiple data are stacked on top of each other. If False (default) multiple data are aranged side by side if histtype is 'bar' or on top of each other if histtype is 'step'.

Returns

n [CArray or list of arrays] The values of the histogram bins. See density and weights for a description of the possible semantics. If input x is an array, then this is an array of length nbins. If input is a sequence arrays [data1, data2,..], then this is a list of arrays with the values of the histograms for each of the arrays in the same order.

bins [CArray] The edges of the bins. Length nbins + 1 (nbins left edges and right edge of last bin). Always a single array even when multiple data sets are passed in.

patches [list or list of lists] Silent list of individual patches used to create the histogram or list of such list if multiple input datasets.

Examples

```
from secml.array import CArray
from secml.figure import CFigure

fig = CFigure(fontsize=14)

# example data
mu = 100  # mean of distribution
sigma = 15  # standard deviation of distribution
x = mu + sigma * CArray.randn((10000,))
num_bins = 50
# the histogram of the data
n, bins, patches = fig.sp.hist(x, num_bins, density=1, facecolor='green',␣
↪alpha=0.5)
# add a 'best fit' line
y = bins.normpdf(mu, sigma)
fig.sp.plot(bins, y, 'r--')
fig.sp.xlabel('Smarts')
fig.sp.ylabel('Probability')
fig.title(r'Histogram of IQ: $\mu=100$, $\sigma=15$')

# Tweak spacing to prevent clipping of ylabel
fig.subplots_adjust(left=0.15)

fig.sp.grid()
fig.show()
```

imshow(*self*, *img*, **args*, ***kwargs*)
Plot image.

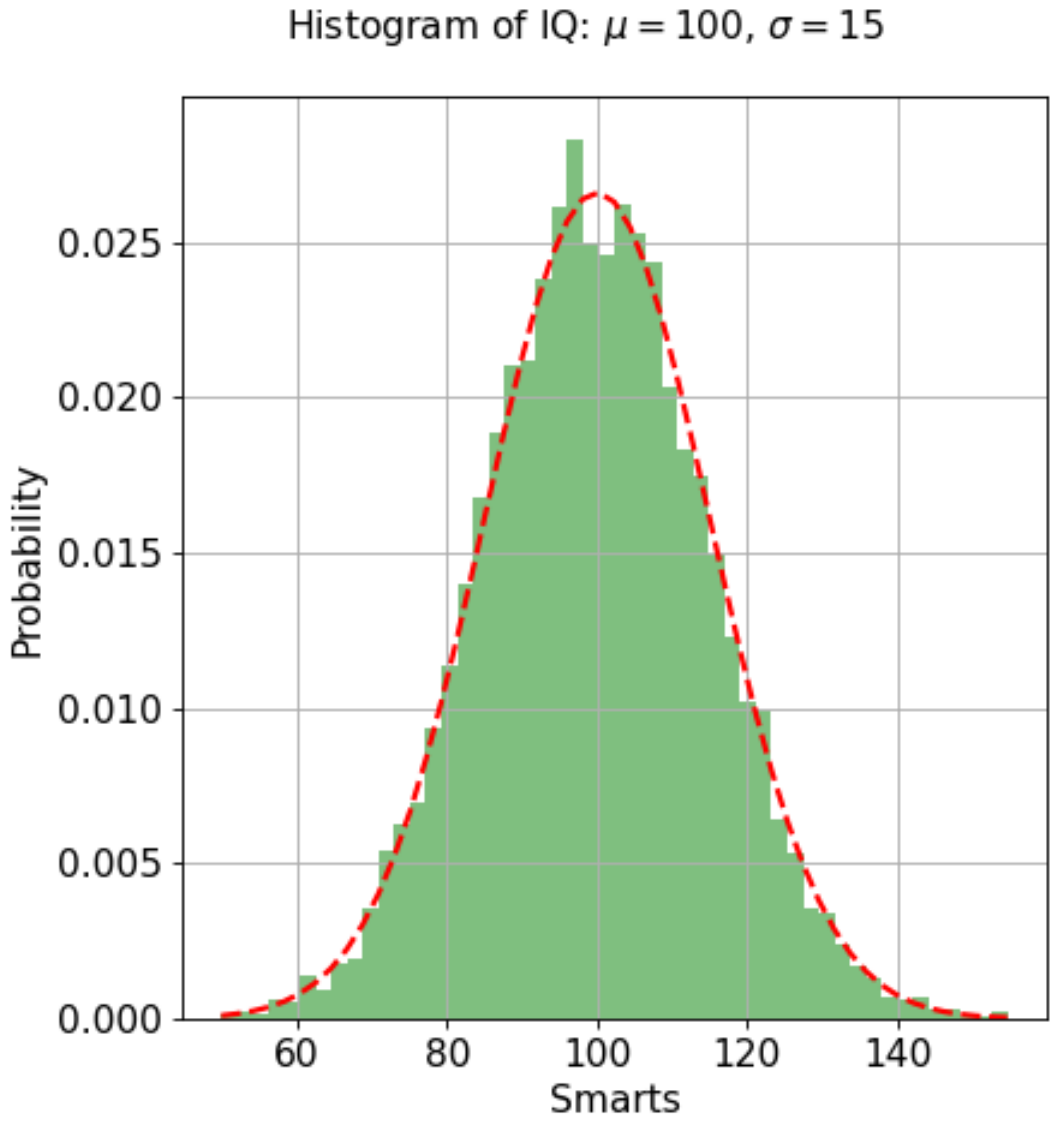
Parameters

img [CArray or PIL.Image.Image] Image to plot.

legend(*self*, **args*, ***kwargs*)
Create legend for plot.

Parameters

loc: integer or string or pair of floats, default: 0


Histogram of IQ: $\mu = 100$, $\sigma = 15$
Probability
0.025
0.020
0.015
0.010
0.005
0.000
60
80
100
120
140
Smarts

Integer	Location
0	‘best’
1	‘upper right’
2	‘upper left’
3	‘lower left’
4	‘lower right’
5	‘right’
6	‘center left’
7	‘center right’
8	‘lower center’
9	‘upper center’
10	‘center’

bbox_to_anchor [tuple of floats] Specify any arbitrary location for the legend in bbox_transform coordinates (default Axes coordinates). For example, to put the legend’s upper right hand corner in the center of the axes the following keywords can be used: loc=’upper right’, bbox_to_anchor=(0.5, 0.5).

ncol [integer] The number of columns that the legend has. Default is 1.

prop [None or dict] The font properties of the legend. If None (default), the current default parameters will be used.

fontsize [int or float or {‘xx-small’, ‘x-small’, ‘small’, ‘medium’, ‘large’, ‘x-large’, ‘xx-large’}] Controls the font size of the legend. If the value is numeric the size will be the absolute font size in points. String values are relative to the current default font size. This argument is only used if prop is not specified.

numpoints [None or int] The number of marker points in the legend when creating a legend entry for a line. Default is None which will take the value from the legend.numpoints default parameter.

scatterpoints [None or int] The number of marker points in the legend when creating a legend entry for a scatter plot. Default is None which will take the value from the legend.scatterpoints default parameter.

scatteryoffsets [iterable of floats] The vertical offset (relative to the font size) for the markers created for a scatter plot legend entry. 0.0 is at the base the legend text, and 1.0 is at the top. To draw all markers at the same height, set to [0.5]. Default [0.375, 0.5, 0.3125].

markerscale [None or int or float] The relative size of legend markers compared with the originally drawn ones. Default is None which will take the value from the legend.markerscale default parameter.

frameon [None or bool] Control whether a frame should be drawn around the legend. Default is None which will take the value from the legend.frameon default parameter.

fancybox [None or bool] Control whether round edges should be enabled around the FancyBboxPatch which makes up the legend’s background. Default is None which will take the value from the legend.fancybox default parameter.

shadow [None or bool] Control whether to draw a shadow behind the legend. Default is None which will take the value from the legend.shadow default parameter.

framealpha [None or float] Control the alpha transparency of the legend’s frame. Default is None which will take the value from the legend.framealpha default parameter.

mode [either between {“expand”, None}] If mode is set to “expand” the legend will be horizontally expanded to fill the axes area (or bbox_to_anchor if defines the legend’s size).

bbox_transform [None or matplotlib.transforms.Transform] The transform for the bounding box (bbox_to_anchor). For a value of None (default) the Axes' transAxes transform will be used.

title [str or None] The legend's title. Default is no title (None).

borderpad [float or None] The fractional whitespace inside the legend border. Measured in font-size units. Default is None which will take the value from the legend.borderpad default parameter.

labelspacing [float or None] The vertical space between the legend entries. Measured in font-size units. Default is None which will take the value from the legend.labelspacing default parameter.

handlelength [float or None] The length of the legend handles. Measured in font-size units. Default is None which will take the value from the legend.handlelength default parameter.

handletextpad [float or None] The pad between the legend handle and text. Measured in font-size units. Default is None which will take the value from the legend.handletextpad default parameter.

borderaxespad [float or None] The pad between the axes and legend border. Measured in font-size units. Default is None which will take the value from the legend.borderaxespad default parameter.

columnspacing [float or None] The spacing between columns. Measured in font-size units. Default is None which will take the value from the legend.columnspacing default parameter.

***args, ****kwargs** Same as *text*.

Examples

```
from secml.array import CArray
from secml.figure import CFigure

X = CArray.linspace(-3.14, 3.14, 256, endpoint=True)
C, S = X.cos(), X.sin()

fig = CFigure(fontsize=14)
fig.sp.plot(X, C, color='red', alpha=0.5, linewidth=1.0, linestyle='-', label=
↪"cosine")
fig.sp.plot(X, S, label="sine")

fig.sp.grid()
fig.sp.legend(loc=0)

fig.show()
```

load_state(*self*, *path*)

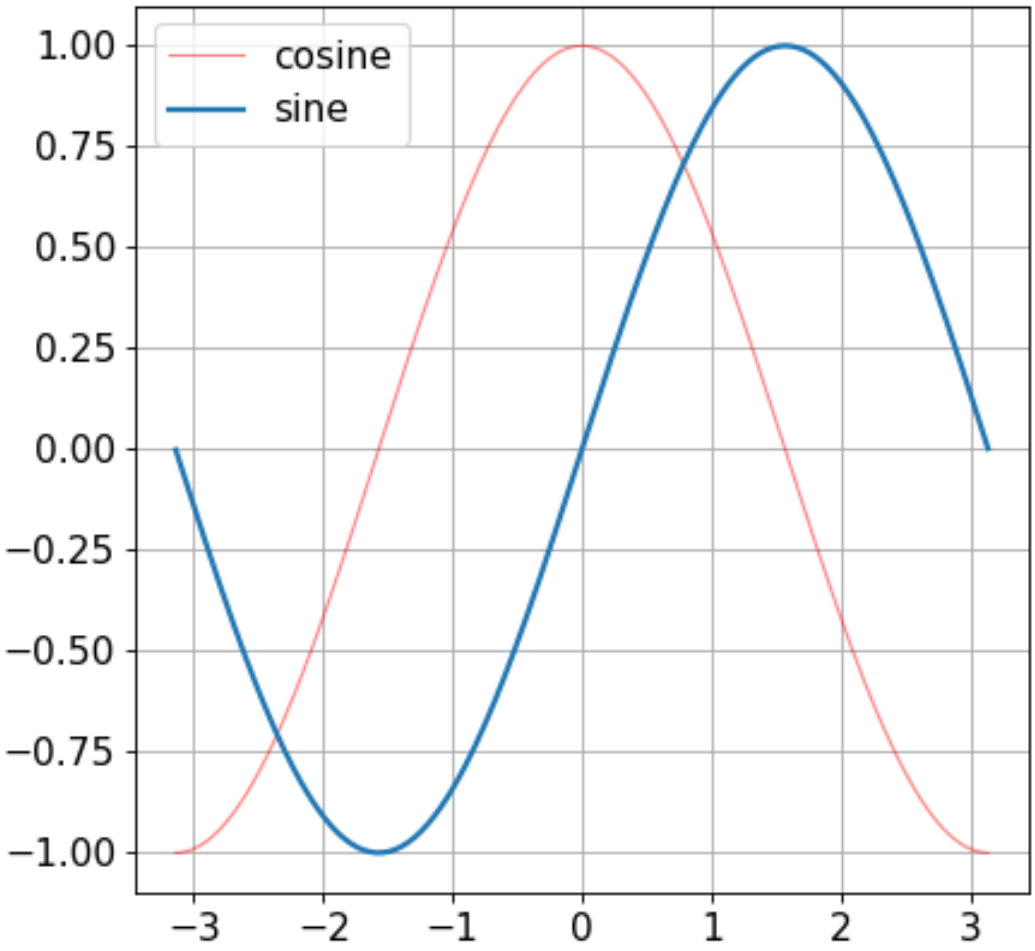
Sets the object state from file.

Parameters

path [str] The full path of the file from which to load the object state.

See also:

set_state Sets the object state using input dictionary.


cosine
sine
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−3
−2
−1
0
1
2
3

loglog(*self*, *x*, *y=None*, **args*, ***kwargs*)
Plot with log scaling on both the x and y axis.

If only one array is given it is supposed to be the y axis data. x axis values are set as index array 0..N-1 .

Parameters

x [list or CArray] x axis values.

y [list or CArray] y axis values.

basex, basey [scalar > 1, default is 10] Base of the x/y logarithm.

subsx, subsy [[None | sequence]] Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9] will place 8 logarithmically spaced minor ticks between each major tick.

nonposx, nonposy [['mask' | 'clip'], default 'mask'.] Non-positive values in x or y can be masked as invalid, or clipped to a very small positive number.

See also:

plot Plot with standard axis.

matshow(*self*, *array*, **args*, ***kwargs*)
Plot an array as a matrix.

Parameters

array [CArray] Array that we want plot as a matrix.

merge(*self*, *sp*)
Merge input subplot to active subplot.

Parameters

sp [CPlot] Subplot to be merged.

property n_lines
Returns the number of lines inside current subplot.

plot(*self*, *x*, *y=None*, **args*, ***kwargs*)
Plot a line.

If only one array is given it is supposed to be the y axis data. x axis values are set as index array 0..N-1 .

Parameters

x [list or CArray] x axis values

y [list or CArray] y axis values

color [str]

Character	Color
'b'	blue
'g'	green
'r'	red
'c'	cyan
'm'	magenta
'y'	yellow
'k'	black
'w'	white

alpha [float, default 1.0] 0.0 for transparent through 1.0 opaque

linestyle [character, default '-'] Can be one into this list : ['-' | '–' | '-.' | ':' | 'None' | ' ' | '']

linewidth [float] 0.0 to 1.0

marker [str]

Character	Marker
'.'	point marker
','	pixel marker
'o'	circle marker
'v'	triangle_down marker
'^'	triangle_up marker
'<'	triangle_left marker
'>'	triangle_right marker
'1'	tri_down marker
'2'	tri_up marker
'3'	tri_left marker
'4'	tri_right marker
's'	square marker
'p'	pentagon marker
'*'	star marker
'h'	hexagon1 marker
'H'	hexagon2 marker
'+'	plus marker
'x'	x marker
'D'	diamond marker
'd'	thin_diamond marker
'|'	vline marker
'_'	hline marker

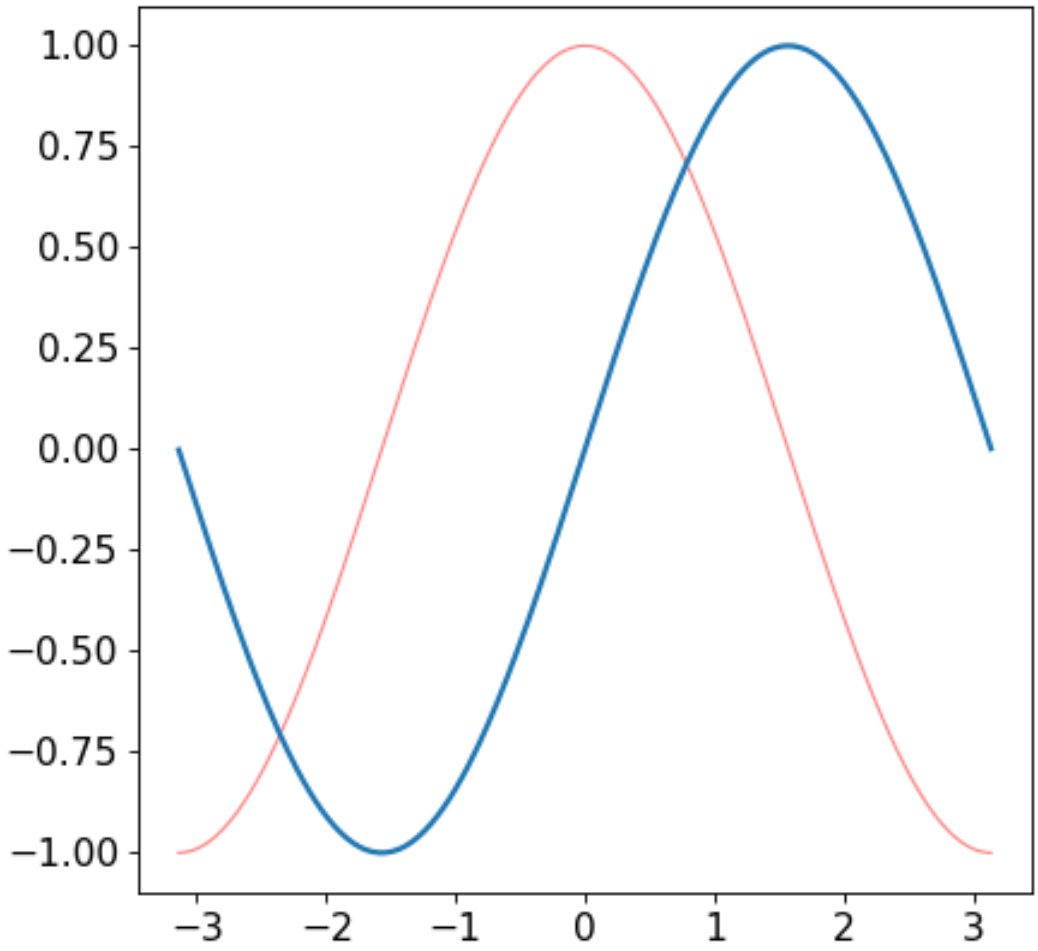
Examples

```
from secml.array import CArray
from secml.figure import CFigure

X = CArray.linspace(-3.14, 3.14, 256, endpoint=True)
C, S = X.cos(), X.sin()

fig = CFigure(fontsize=14)

fig.sp.plot(X, C, color='red', alpha=0.5, linewidth=1.0, linestyle='-')
fig.sp.plot(X, S)

fig.show()
```

plot_path(*self*, *path*, *path_style='-'*, *path_width=1.5*, *path_color='k'*, *straight=False*, *start_style='h'*, *start_facecolor='r'*, *start_edgecolor='k'*, *start_edgewidth=1*, *final_style='*'*, *final_facecolor='g'*, *final_edgecolor='k'*, *final_edgewidth=1*)

Plot a path traversed by a point.

By default, path is drawn in solid black, start point is drawn with a red star and the end point is drawn with a green asterisk.


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−3
−2
−1
0
1
2
3

Parameters

path [CArray] Every row contain one point coordinate.

path_style [str] Style for the path line. Default solid (-).

path_width [int] Width of path line. Default 1.5.

path_color [str] Color for the path line. Default black (k).

straight [bool, default False] If True, path will be plotted straight between start and end point.

start_style [str] Style for the start point. Default an hexagon (h).

start_facecolor [str] Color for the start point. Default red (r).

start_edgecolor [str] Color for the edge of the start point marker. Default black (k).

start_edgewidth [scalar] Width of the edge for the start point. Default 1.

final_style [str] Style for the end point. Default a star (*).

final_facecolor [str] Color for the end point. Default red (g).

final_edgecolor [str] Color for the edge of the final point marker. Default black (k).

final_edgewidth [scalar] Width of the edge for the end point. Default 1.

Examples

```
from secml.array import CArray
from secml.figure import CFigure

fig = CFigure(fontsize=14)
fig.sp.title("5-points path")

path = CArray([[2, 2], [3, 2], [4, 7], [5, 4], [1, 3]])

fig.sp.plot_path(path)

fig.sp.xlim(0, 6)
fig.sp.ylim(1, 8)

fig.show()
```

quiver (*self*, *U*, *V*, *X=None*, *Y=None*, *color='k'*, *linestyle='-'*, *linewidth=1.0*, *alpha=1.0*)

A quiver plot displays velocity vectors as arrows with components (u,v) at the points (x,y).

For example, the first vector is defined by components u(1), v(1) and is displayed at the point x(1), y(1).

quiver(x,y,u,v) plots vectors as arrows at the coordinates specified in each corresponding pair of elements in x and y.

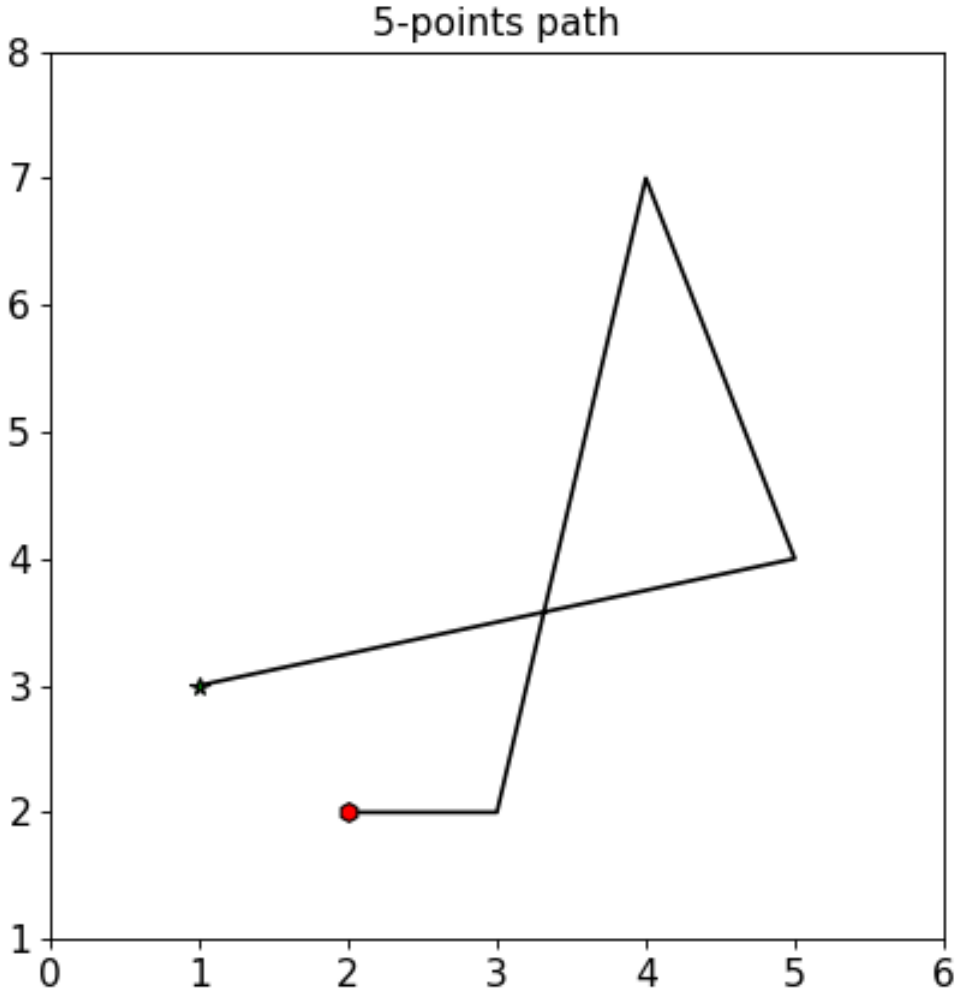
quiver(u,v) draws vectors specified by u and v at equally spaced points in the x-y plane.

Parameters

U, V: scalar or CArray Give the x and y components of the arrow vectors.

X, Y: scalar or CArray, optional The x and y coordinates of the arrow locations (default is tail of arrow; see pivot kwarg)

color : Color of the gradient directions.


5-points path
8
7
6
5
4
3
2
1
0
1
2
3
4
5
6

linestyle [str] [‘solid’ | ‘dashed’, ‘dashdot’, ‘dotted’ | (offset, on-off-dash-seq) | ‘-‘ | ‘–’ | ‘-.’ | ‘:’ | ‘None’ | ‘ ‘ | ‘’]

linewidth [float] Width of the line.

alpha [float] Transparency factor of the directions.

save_state(*self*, *path*)
Store the object state to file.

Parameters

path [str] Path of the file where to store object state.

Returns

str The full path of the stored object.

See also:

get_state Returns the object state dictionary.

scatter(*self*, *x*, *y*, *s=20*, *c='b'*, **args*, ***kwargs*)
Scatter plot of x vs y.

Parameters

x, y [list or CArray] Input data. Both object must have the same size.

s [scalar or shape (n,), optional, default: 20] size in points^2.

c [color or sequence of color, optional, default ‘b’] c can be a single color format string, or a sequence of color specifications of length N, or a sequence of numbers with the same shape of x,y to be mapped to colors using the cmap and norm specified via kwargs (see below). Note that c should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. c can be a 2-D array in which the rows are RGB or RGBA, however.

marker [MarkerStyle, optional, default: ‘o’] See markers for more information on the different styles of markers scatter supports.

cmap [Colormap, optional, default: None] A Colormap instance or registered name. cmap is only used if c is an array of floats. If None, default parameter image.cmap is used.

norm [Normalize, optional, default: None] A Normalize instance is used to scale luminance data to 0, 1. norm is only used if c is an array of floats.

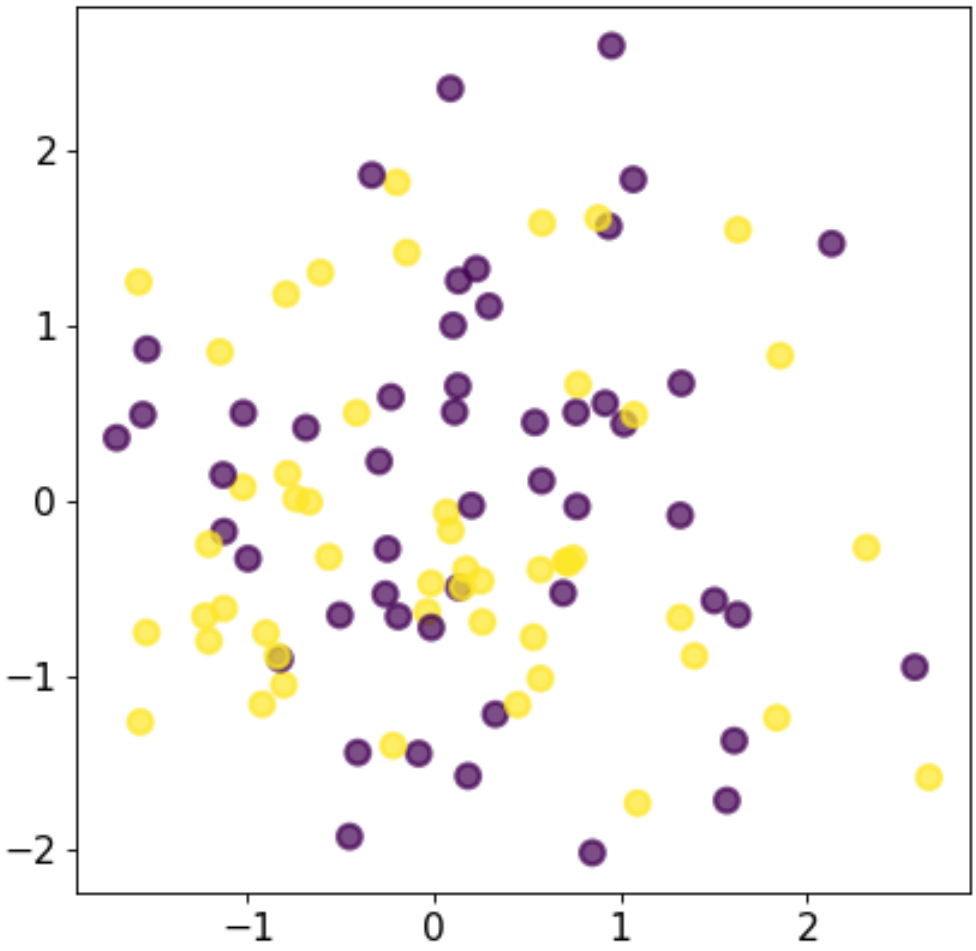
vmin, vmax [scalar, optional, default: None] vmin and vmax are used in conjunction with norm to normalize luminance data. If either are None, the min and max of the color array is used. Note if you pass a norm instance, your settings for vmin and vmax will be ignored.

alpha [scalar, optional, default: None] The alpha blending value, between 0 (transparent) and 1 (opaque)

linewidths [scalar or array_like, optional, default: None] If None, defaults to (lines.linewidth,). Note that this is a tuple, and if you set the linewidths argument you must set it as a sequence of float.

Examples

```
from secml.data.loader import CDLRandom
from secml.figure import CFigure

dataset = CDLRandom().load()

fig = CFigure(fontsize=14)
fig.sp.scatter(dataset.X[:, 0].ravel(),
               dataset.X[:, 1].ravel(),
               s=75, c=dataset.Y, alpha=.7)

fig.show()
```



semilogx (*self*, *x*, *y=None*, **args*, ***kwargs*)

Plot with log scaling on the x axis.

If only one array is given it is supposed to be the y axis data. x axis values are set as index array 0..N-1 .

Parameters

x [list or CArray] x axis values

y [list or CArray] y axis values

basex [scalar > 1, default is 10] Base of the x logarithm

subsx [[None | sequence]] Where to place the subticks between each major tick. Sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9] will place 8 logarithmically spaced minor ticks between each major tick.

nonposx [['mask' | 'clip'], default 'mask'] Non-positive values in x can be masked as invalid, or clipped to a very small positive number

See also:

plot Plot with standard axis.

Examples

```
from secml.array import CArray
from secml.figure import CFigure

fig = CFigure(fontsize=14)

t = CArray.arange(0.01, 20.0, 0.01)
fig.sp.semilogx(t, (2 * 3.14 * t).sin())

fig.sp.grid()
fig.sp.title('semilogx')

fig.show()
```

semilogy (*self*, *x*, *y=None*, **args*, ***kwargs*)

Plot with log scaling on the y axis.

If only one array is given it is supposed to be the y axis data. x axis values are set as index array 0..N-1 .

Parameters

x [list or CArray] x axis values.

y [list or CArray] y axis values.

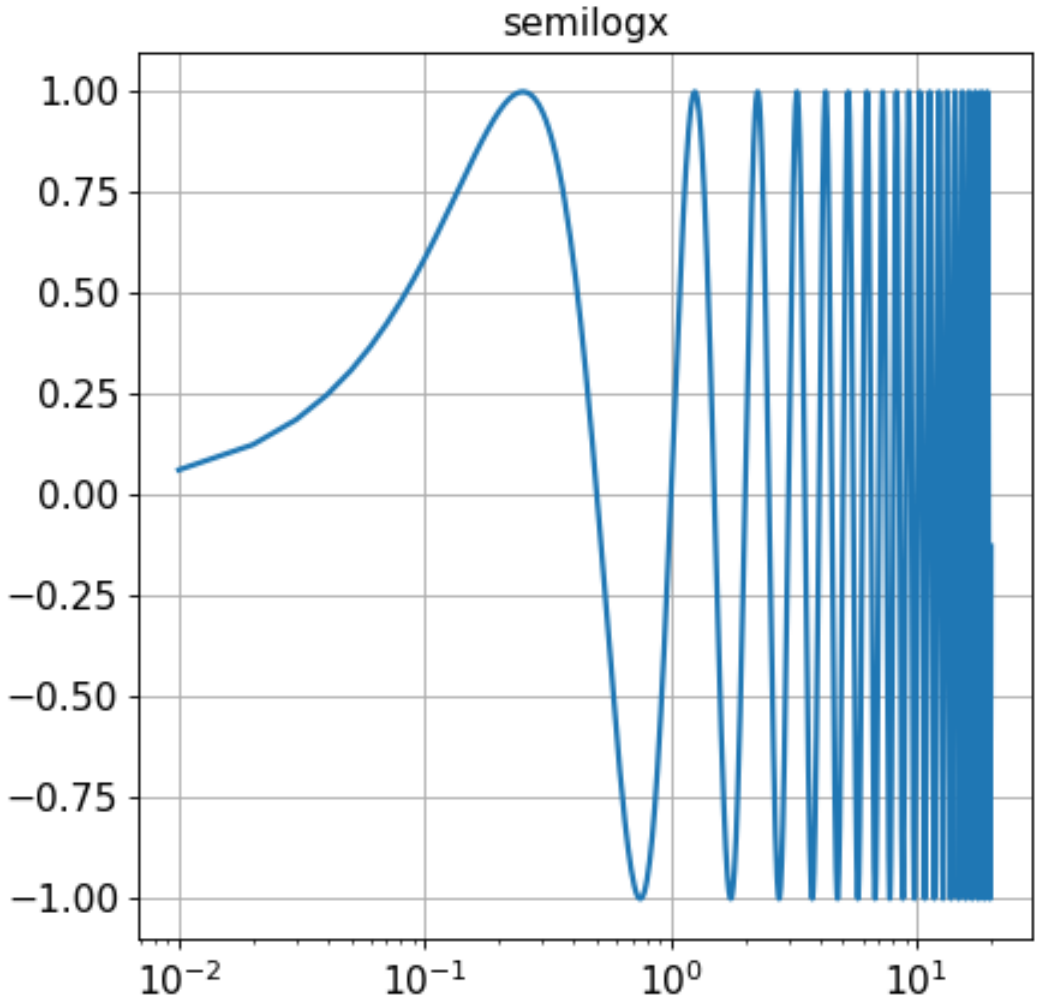
basey [scalar > 1, default is 10] Base of the y logarithm

subsy [[None | sequence], default None] Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: [2, 3, 4, 5, 6, 7, 8, 9] will place 8 logarithmically spaced minor ticks between each major tick.

nonposy [['mask' | 'clip'], default 'mask'] Non-positive values in x can be masked as invalid, or clipped to a very small positive number.

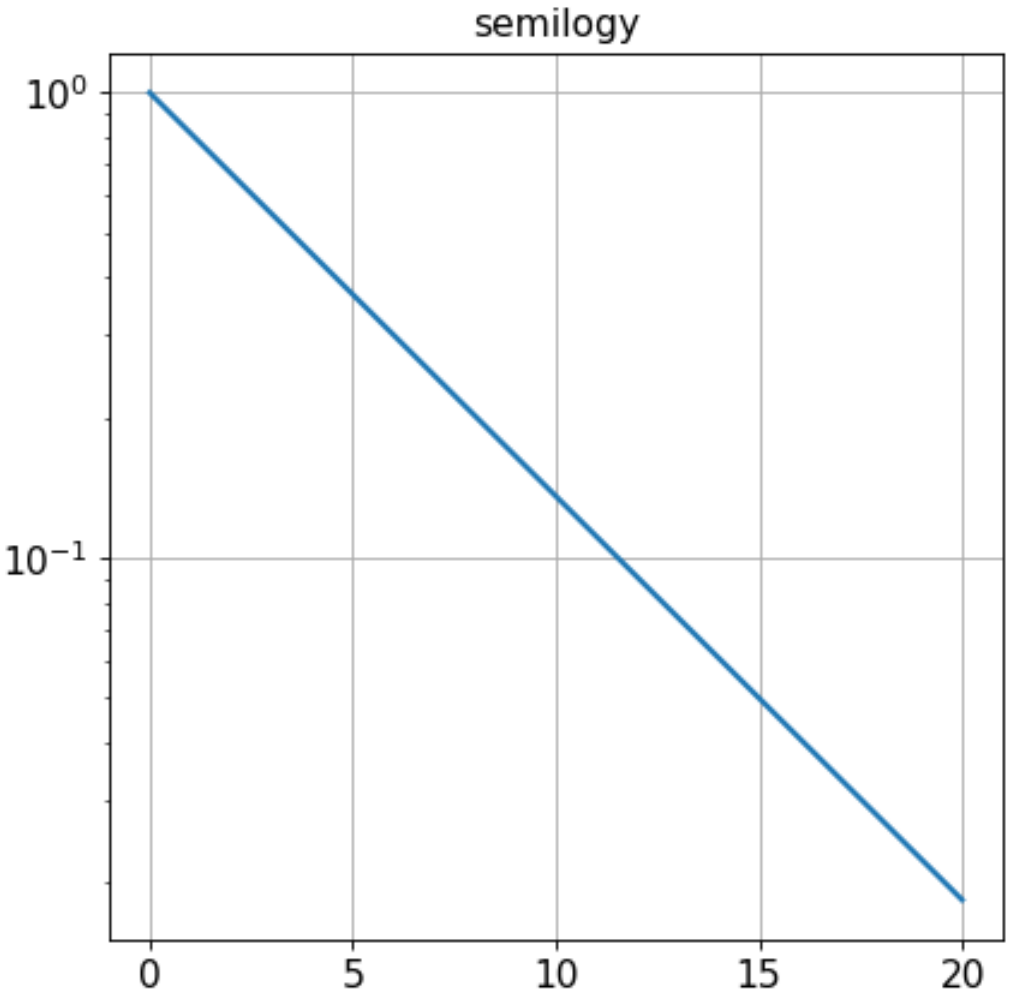
See also:

plot Plot with standard axis.


semilogx
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
10^{-2}
10^{-1}
10^{0}
10^{1}

Examples

```
from secml.array import CArray
from secml.figure import CFigure

fig = CFigure(fontsize=14)

t = CArray.arange(0.01, 20.0, 0.01)
fig.sp.semilogy(t, (-t / 5.0).exp())

fig.sp.title('semilogy')
fig.sp.grid()
fig.show()
```



set (*self*, *param_name*, *param_value*, *copy=False*)
Set a parameter of the class.

Only writable attributes of the class, i.e. PUBLIC or READ/WRITE, can be set.

The following checks are performed before setting:

- if *param_name* is an attribute of current class, set directly;
- **else, iterate over __dict__ and look for a class attribute** having the desired parameter as an attribute;
- **else, if attribute is not found on the 2nd level,** raise AttributeError.

If possible, a reference to the attribute to set is assigned. Use *copy=True* to always make a deepcopy before set.

Parameters

param_name [str] Name of the parameter to set.

param_value [any] Value to set for the parameter.

copy [bool] By default (False) a reference to the parameter to assign is set. If True or a reference cannot be extracted, a deepcopy of the parameter value is done first.

set_axisbelow(*self*, *axisbelow=True*)
Set axis ticks and gridlines below most artists.

set_state(*self*, *state_dict*, *copy=False*)
Sets the object state using input dictionary.

Only readable attributes of the class, i.e. PUBLIC or READ/WRITE or READ ONLY, can be set.

If possible, a reference to the attribute to set is assigned. Use *copy=True* to always make a deepcopy before set.

Parameters

state_dict [dict] Dictionary containing the state of the object.

copy [bool, optional] By default (False) a reference to the attribute to assign is set. If True or a reference cannot be extracted, a deepcopy of the attribute is done first.

text(*self*, **args*, ***kwargs*)
Create a Text instance at x, y with string text.

Parameters

Any of the following keyword arguments is supported.

Text properties:

Property	Description
alpha	float (0.0 transparent through 1.0 opaque)
animated	[True | False]
background-color	one of possible color
bbox	rectangle prop dict
color	one of possible color
family or font-family or font-name or name	[FONTNAME | 'serif' | 'sans-serif' | 'cursive' | 'fantasy' | 'monospace']
horizonta-lalignment or ha	['center' | 'right' | 'left']
label	string or anything printable with '%s' conversion.
linespacing	float (multiple of font size)
position	(x,y)
rasterized	[True | False | None]
rotation	[angle in degrees | 'vertical' | 'horizontal']
size or fontsize	[size in points | 'xx-small' | 'x-small' | 'small' | 'medium' | 'large' | 'x-large' | 'xx-large']
stretch or fontstretch	[a numeric value in range 0-1000 | 'ultra-condensed' | 'extra-condensed' | 'condensed' | 'semi-condensed' | 'normal' | 'semi-expanded' | 'expanded' | 'extra-expanded' | 'ultra-expanded']
style or fontstyle	['normal' | 'italic' | 'oblique']
text	string or anything printable with '%s' conversion.
verticalalign-ment or va or ma	['center' | 'top' | 'bottom' | 'baseline']
visible	[True | False]
weight or fontweight	[a numeric value in range 0-1000 | 'ultralight' | 'light' | 'normal' | 'regular' | 'book' | 'medium' | 'roman' | 'semibold' | 'demibold' | 'demi' | 'bold' | 'heavy' | 'extra bold' | 'black']
x	float, x position of the text.
y	float. y position of the text.
zorder	any number, objects with lower zorder values are drawn first.

Font properties:

Property	Description
family	(font name or font family) es: 'serif', 'sans-serif', 'cursive', 'fantasy', or 'monospace'
style	either between 'normal', 'italic' or 'oblique'
variant	'normal' or 'small-caps'
stretch	A numeric value in the range 0-1000 or one of 'ultra-condensed', 'extra-condensed', 'condensed', 'semi-condensed', 'normal', 'semi-expanded', 'expanded', 'extra-expanded' or 'ultra-expanded'
weight	A numeric value in the range 0-1000 or one of 'ultralight', 'light', 'normal', 'regular', 'book', 'medium', 'roman', 'semibold', 'demibold', 'demi', 'bold', 'heavy', 'extra bold', 'black'
size	Either an relative value of 'xx-small', 'x-small', 'small', 'medium', 'large', 'x-large', 'xx-large' or an absolute font size, e.g., 12

tick_params (*self*, **args*, ***kwargs*)

Change the appearance of ticks and tick labels.

Parameters

axis [['x' | 'y' | 'both']] Axis on which to operate; default is 'both'.

reset [[True | False]] Default False. If True, set all parameters to defaults before processing other keyword arguments.

which [['major' | 'minor' | 'both']] Default is 'major'; apply arguments to which ticks.

direction [['in' | 'out' | 'inout']] Puts ticks inside the axes, outside the axes, or both.

length [int] Tick length in points.

width [int] Tick width in points.

color [str] Tick color; accepts any mpl color spec.

pad [int] Distance in points between tick and label.

labelsize [int, str] Tick label font size in points or as a string (e.g., 'large').

labelcolor [str] Tick label color; mpl color spec.

colors [str] Changes the tick color and the label color to the same value: mpl color spec.

bottom, top, left, right [bool, optional] Controls whether to draw the respective ticks.

labelbottom, labeltop, labelleft, labelright [bool, optional] Controls whether to draw the respective tick labels.

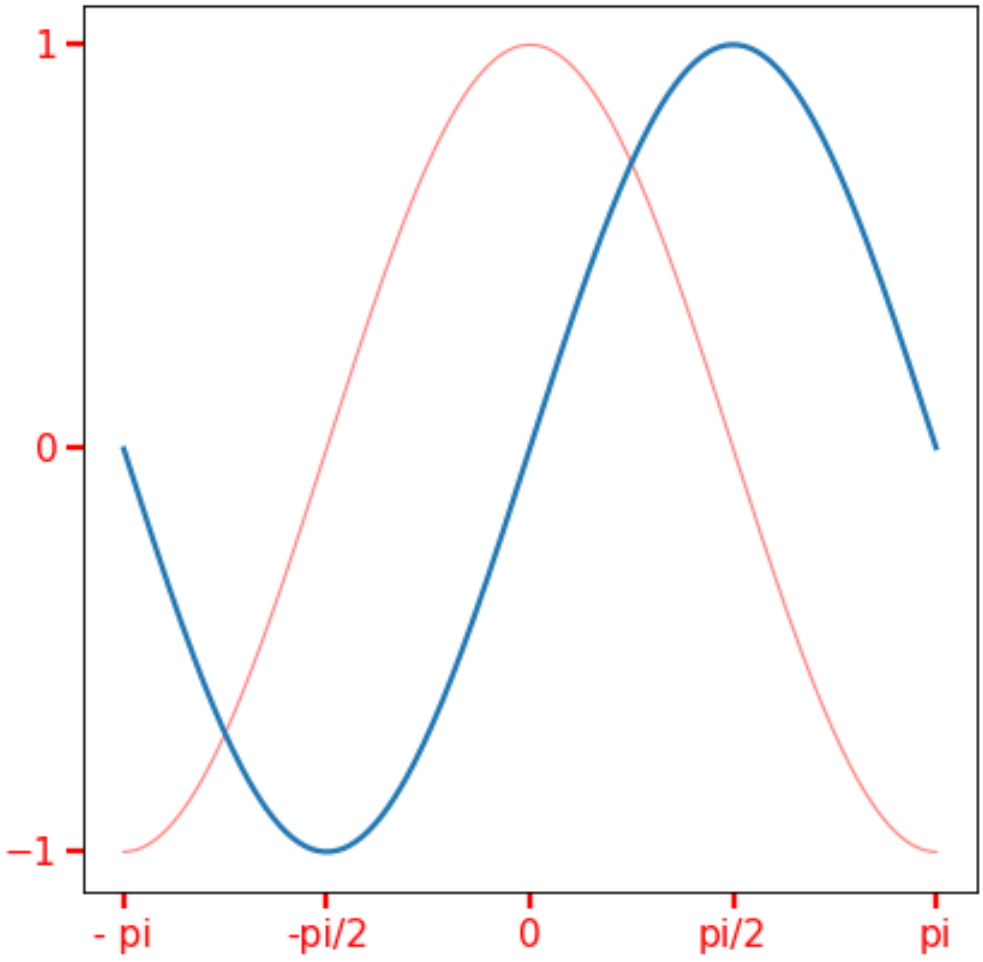
Examples

```
from secml.array import CArray
from secml.figure import CFigure
from secml.core.constants import pi

X = CArray.linspace(-3.14, 3.14, 256, endpoint=True)
C, S = X.cos(), X.sin()

fig = CFigure(fontsize=14)
```

(continues on next page)

(continued from previous page)

```
fig.sp.plot(X, C, color='red', alpha=0.5, linewidth=1.0, linestyle='-')
fig.sp.plot(X, S)

fig.sp.xticks(CArray([-pi, -pi / 2, 0, pi / 2, pi]))
fig.sp.xticklabels(CArray(["- pi", "-pi/2", "0", "pi/2", "pi"]))
fig.sp.tick_params(direction='out', length=6, width=2, colors='r',␣
↪right=False)
fig.sp.yticks(CArray([-1, 0, +1]))

fig.show()
```



title (*self*, *text*, **args*, ***kwargs*)
Set a title for subplot.

xlabel (*self*, *label*, **args*, ***kwargs*)
Set a label for the x axis.

Parameters

label [string] Label's text.

***args, **kwargs** Same as *text* method.

Examples

```
from secml.array import CArray
from secml.figure import CFigure

X = CArray.linspace(-3.14, 3.14, 256, endpoint=True)
C, S = X.cos(), X.sin()

fig = CFigure(fontsize=14)

fig.sp.plot(X, C, color='red', alpha=0.5, linewidth=1.0, linestyle='-')
fig.sp.plot(X, S)

fig.sp.xlabel("x", color='r', fontsize=10)

fig.show()
```

xlim(*self*, *bottom=None*, *top=None*)

Set axes x limits.

Parameters

bottom [scalar] Starting value for the x axis.

top [scalar] Ending value for the x axis.

Examples

```
from secml.array import CArray
from secml.figure import CFigure

X = CArray.linspace(-3.14, 3.14, 256, endpoint=True)
C, S = X.cos(), X.sin()

fig = CFigure(fontsize=14)

fig.sp.plot(X, C, color='red', alpha=0.5, linewidth=1.0, linestyle='-')
fig.sp.plot(X, S)

fig.sp.xlim(-3, 3)

fig.show()
```

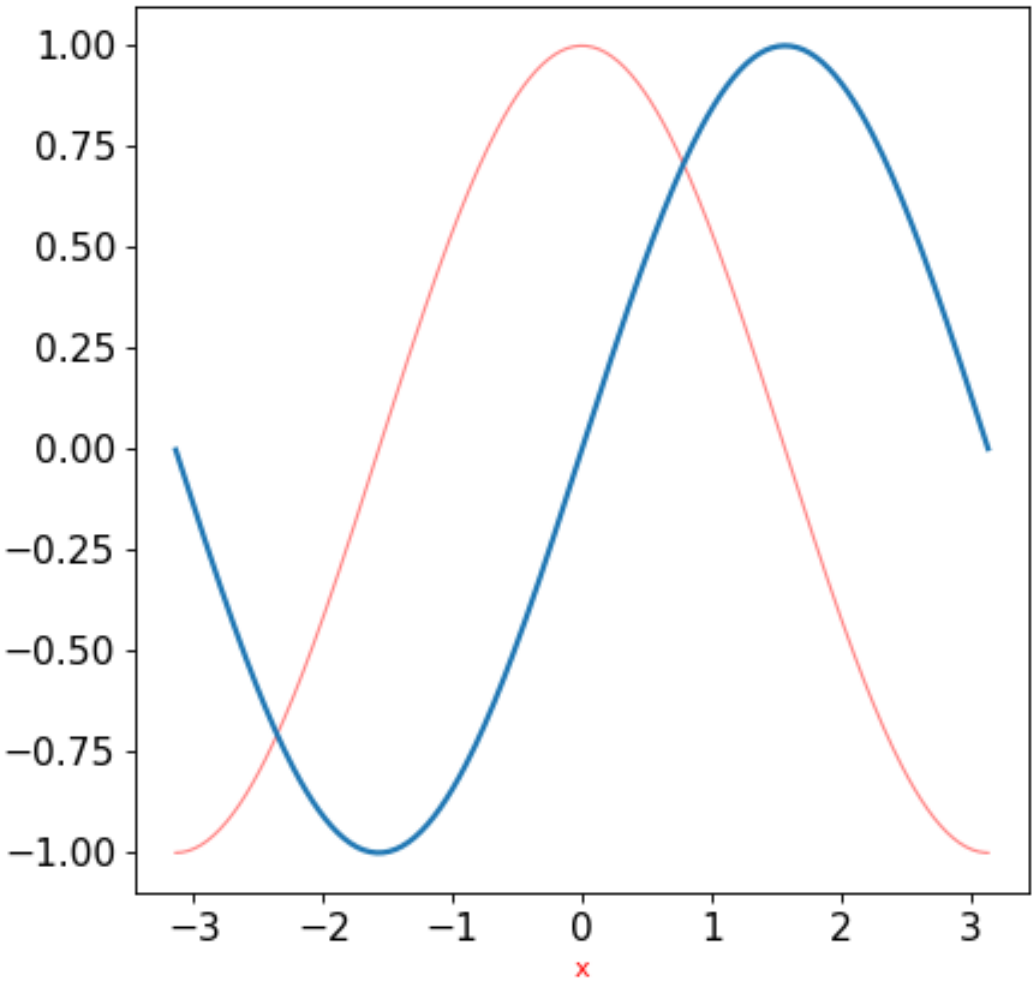
xscale(*self*, *scale_type*, *nonposx='mask'*, *basex=10*, ***kwargs*)

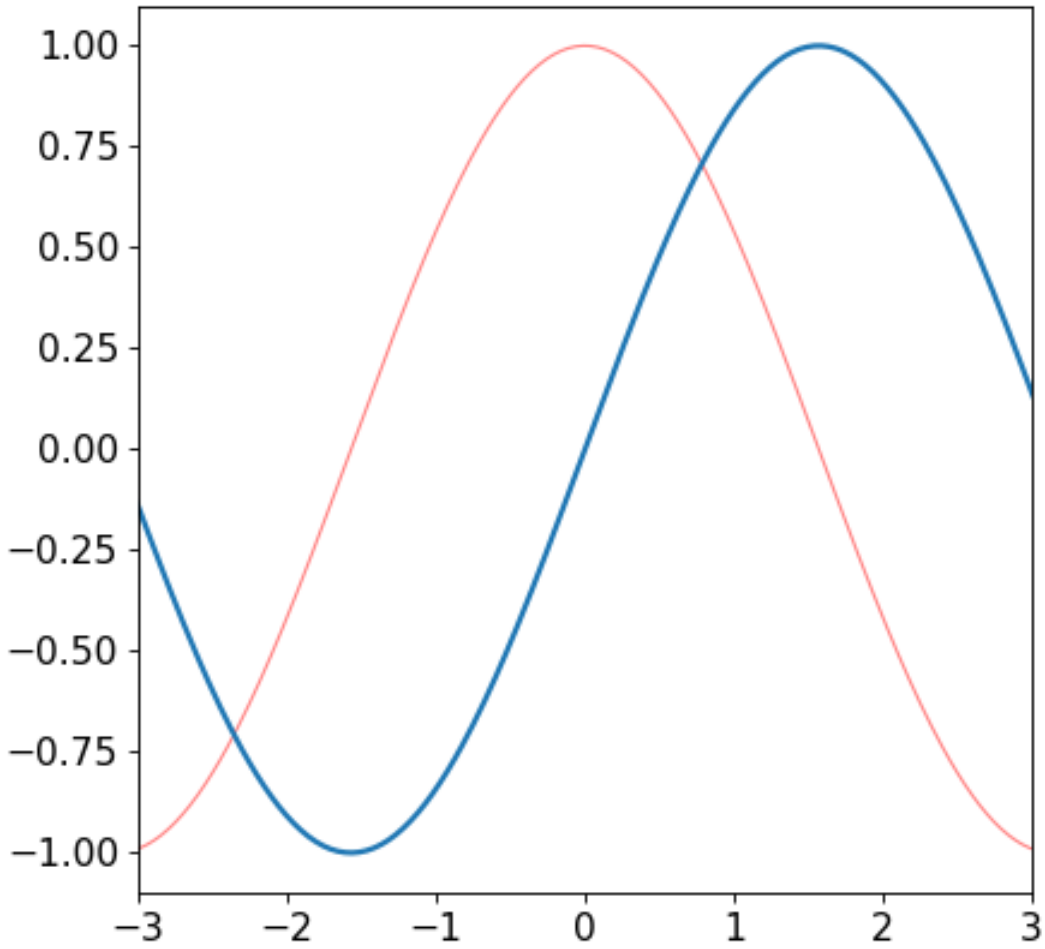
Set scale for x axis.

Parameters

scale_type [{'linear', 'log', 'symlog', 'logit'}] Scale for x axis. Default 'linear'.

nonposx: ['mask' | 'clip'], default 'mask' Non-positive values in x can be masked as invalid, or clipped to a very small positive number.


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−3
−2
−1
0
1
2
3
x


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−3
−2
−1
0
1
2
3

basex [int] The base of the logarithm, must be higger than 1.

xticklabels (*self*, *labels*, **args*, ***kwargs*)
Set the xtick labels.

Parameters

labels [list or CArray of string] Xtick labels.

***args, **kwargs** Same as *text* method.

Examples

```
from secml.array import CArray
from secml.figure import CFigure

X = CArray.linspace(-3.14, 3.14, 256, endpoint=True)
C, S = X.cos(), X.sin()

fig = CFigure(fontsize=14)

fig.sp.plot(X, C, color='red', alpha=0.5, linewidth=1.0, linestyle='-')
fig.sp.plot(X, S)

fig.sp.xticks(CArray([-3.14, -3.14 / 2, 0, 3.14 / 2, 3.14]))
fig.sp.xticklabels(["- pi", "-pi/2", "0", "pi/2", "pi"])

fig.sp.yticks(CArray([-1, 0, +1]))

fig.show()
```

xticks (*self*, *location_array*, **args*, ***kwargs*)
Set the x-tick locations and labels.

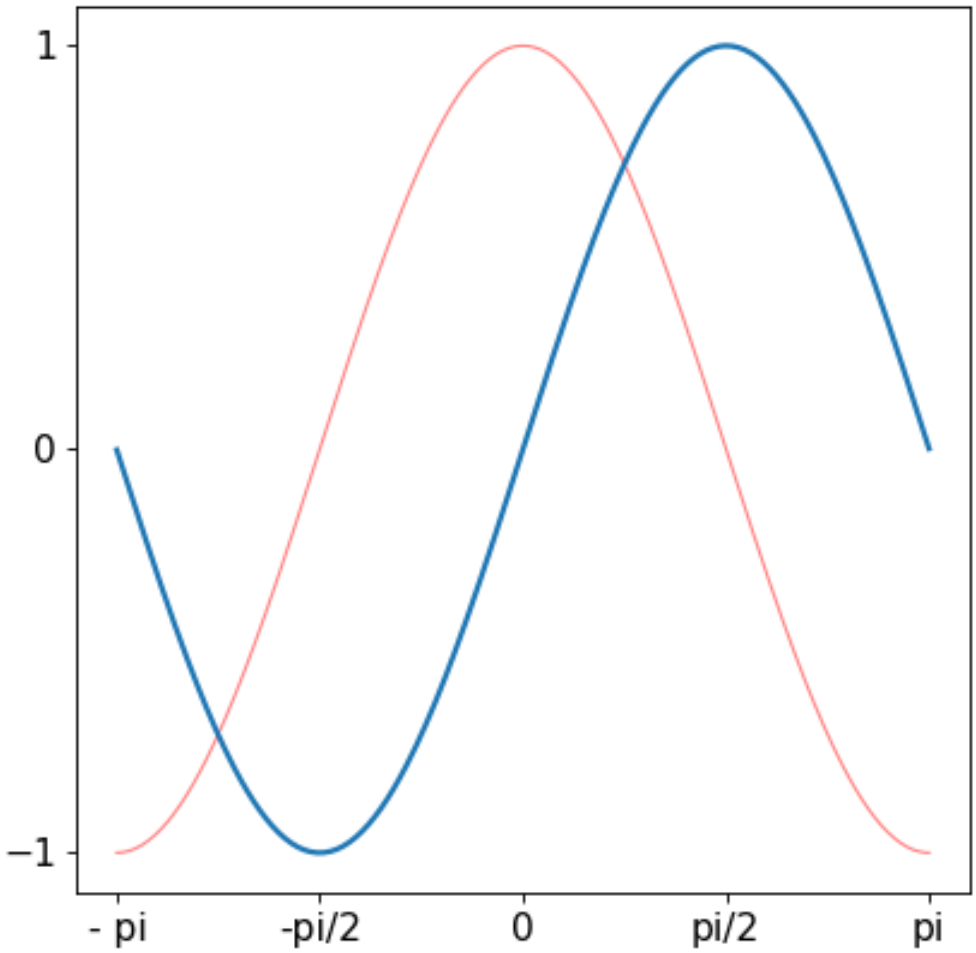
Parameters

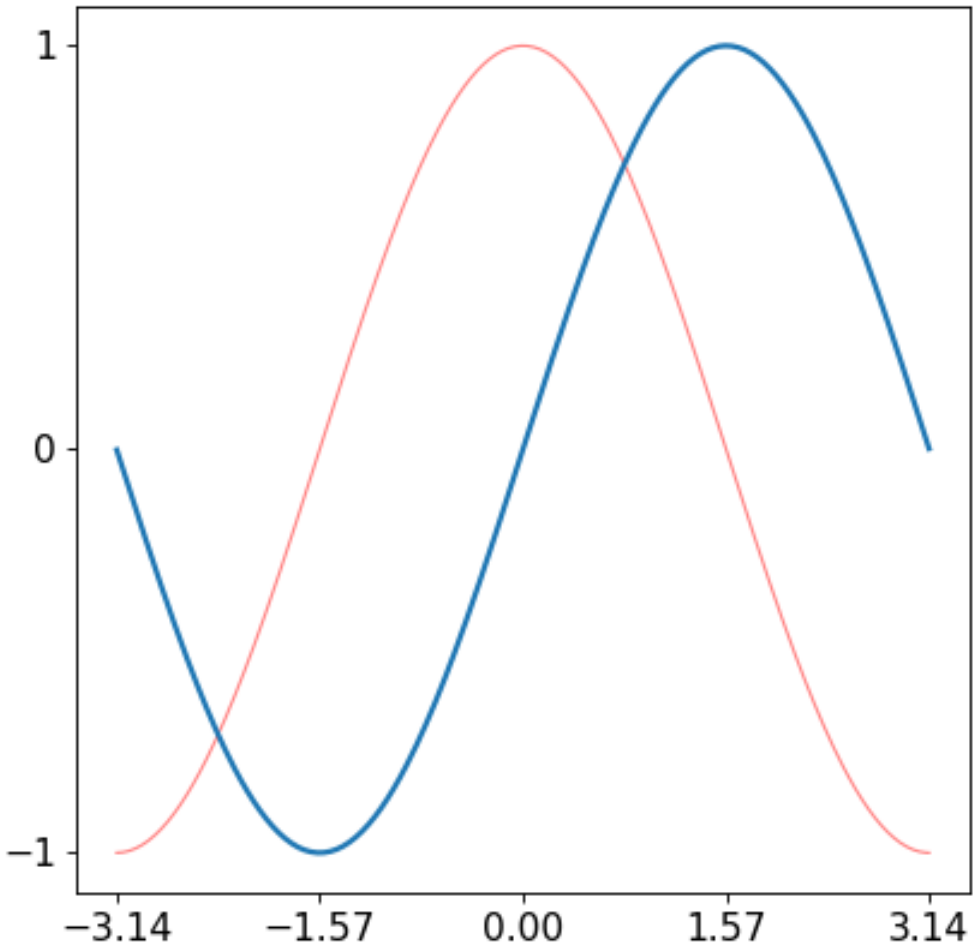
location_array [CArray or list] Contain ticks location.

***args, **kwargs** Same as *text* method.

Examples

```
from secml.array import CArray
from secml.figure import CFigure

X = CArray.linspace(-3.14, 3.14, 256, endpoint=True)
C, S = X.cos(), X.sin()

fig = CFigure(fontsize=14)

fig.sp.plot(X, C, color='red', alpha=0.5, linewidth=1.0, linestyle='-')
fig.sp.plot(X, S)

fig.sp.xticks(CArray([-3.14, -3.14 / 2, 0, 3.14 / 2, 3.14]))
fig.sp.yticks(CArray([-1, 0, +1]))

fig.show()
```


1
0
−1
- pi
-pi/2
0
pi/2
pi


1
0
−1
−3.14
−1.57
0.00
1.57
3.14

ylabel (*self*, *label*, **args*, ***kwargs*)
Set a label for the y axis

Parameters

label [string] Label's text.

***args, **kwargs** Same as *text* method.

See also:

xlabel Set a label for the x axis.

ylim (*self*, *bottom=None*, *top=None*)
Set axes y limits.

Parameters

bottom [scalar] Starting value for the y axis.

top [scalar] Ending value for the y axis.

See also:

xlim Set x axis limits.

yscale (*self*, *scale_type*, *nonposy='mask'*, *basey=10*, ***kwargs*)
Set scale for y axis.

Parameters

scale_type [{'linear', 'log', 'symlog', 'logit'}] Scale for y axis. Default 'linear'.

nonposy: ['mask' | 'clip'], default 'mask' Non-positive values in y can be masked as invalid, or clipped to a very small positive number.

basey [int] The base of the logarithm, must be higger than 1.

yticklabels (*self*, *labels*, **args*, ***kwargs*)
Set the ytick labels.

Parameters

labels [list or CArray of string] Xtick labels.

***args, **kwargs** Same as *text* method.

See also:

xticklabels Set the xtick labels.

yticks (*self*, *location_array*, **args*, ***kwargs*)
Set the y-tick locations and labels.

Parameters

location_array [CArray or list] Contain ticks location.

***args, **kwargs** Same as *text* method.

See also:

xticks Set the x-tick locations and labels.

class secml.figure._plots.c_plot_classifier.**CPlotClassifier**(*sp*, *default_params*)

Plot a classifier.

Custom plotting parameters can be specified.

Currently parameters default:

- grid: False.

See also:

CPlot basic subplot functions.

CFigure creates and handle figures.

Attributes

class_type Defines class type.

logger Logger for current object.

n_lines Returns the number of lines inside current subplot.

verbose Verbosity level of logger output.

Methods

apply_params_clf(self)	Apply defined parameters to active subplot.
apply_params_fun(self)	Apply defined parameters to active subplot.
bar(self, left, height[, width, bottom])	Bar plot.
barh(self, bottom, width[, height, left])	Horizontal bar plot.
boxplot(self, x[, notch, sym, vert, whis, ...])	Make a box and whisker plot.
clabel(self, contour, *args, **kwargs)	Label a contour plot.
colorbar(self, mappable[, ticks])	Add colorbar to plot.
contour(self, x, y, z, *args, **kwargs)	Draw contour lines of a function.
contourf(self, x, y, z, *args, **kwargs)	Draw filled contour of a function.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
errorbar(self, x, y[, xerr, yerr])	Plot with error deltas in yerr and xerr.
fill_between(self, x, y1[, y2, where, ...])	Fill the area between two horizontal curves.
get_class_from_type(class_type)	Return the class associated with input type.
get_legend(self)	Returns the handler of current subplot legend.
get_legend_handles_labels(self)	Return handles and labels for legend contained by the subplot.
get_lines(self)	Return a list of lines contained by the subplot.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
get_xticks_idx(self, xticks)	Returns the position of markers to plot.
grid(self[, grid_on, axis])	Draw grid for current plot.
hist(self, x, *args, **kwargs)	Plot a histogram.
imshow(self, img, *args, **kwargs)	Plot image.
legend(self, *args, **kwargs)	Create legend for plot.

Continued on next page

Table 156 – continued from previous page

list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
loglog(self, x[, y])	Plot with log scaling on both the x and y axis.
matshow(self, array, *args, **kwargs)	Plot an array as a matrix.
merge(self, sp)	Merge input subplot to active subplot.
plot(self, x[, y])	Plot a line.
plot_decision_regions(self, clf[, ...])	Plot decision boundaries and regions for the given classifier.
plot_fgrads(self, gradf[, n_grid_points, ...])	Plot function gradient directions.
plot_fun(self, func[, multipoint, ...])	Plot a function (used for decision functions or boundaries).
plot_path(self, path[, path_style, ...])	Plot a path traversed by a point.
quiver(self, U, V[, X, Y, color, linestyle, ...])	A quiver plot displays velocity vectors as arrows with components (u,v) at the points (x,y).
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
scatter(self, x, y[, s, c])	Scatter plot of x vs y.
semilogx(self, x[, y])	Plot with log scaling on the x axis.
semilogy(self, x[, y])	Plot with log scaling on the y axis.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_axisbelow(self[, axisbelow])	Set axis ticks and gridlines below most artists.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
text(self, *args, **kwargs)	Create a Text instance at x, y with string text.
tick_params(self, *args, **kwargs)	Change the appearance of ticks and tick labels.
timed([msg])	Timer decorator.
title(self, text, *args, **kwargs)	Set a title for subplot.
xlabel(self, label, *args, **kwargs)	Set a label for the x axis.
xlim(self[, bottom, top])	Set axes x limits.
xscale(self, scale_type[, nonposx, basex])	Set scale for x axis.
xticklabels(self, labels, *args, **kwargs)	Set the xtick labels.
xticks(self, location_array, *args, **kwargs)	Set the x-tick locations and labels.
ylabel(self, label, *args, **kwargs)	Set a label for the y axis
ylim(self[, bottom, top])	Set axes y limits.
yscale(self, scale_type[, nonposy, basey])	Set scale for y axis.
yticklabels(self, labels, *args, **kwargs)	Set the ytick labels.
yticks(self, location_array, *args, **kwargs)	Set the y-tick locations and labels.

apply_params_clf(*self*)
Apply defined parameters to active subplot.

plot_decision_regions(*self*, *clf*, *plot_background=True*, *levels=None*, *grid_limits=None*, *n_grid_points=30*, *cmap=None*)
Plot decision boundaries and regions for the given classifier.

Parameters

clf [CClassifier] Classifier which decision function should be plotted.

plot_background [bool, optional] Specifies whether to color the decision regions. Default True. in the background using a colorbar.

levels [list or None, optional] List of levels to be plotted. If None, CArray.arange(0.5, clf.n_classes) will be plotted.

grid_limits [list of tuple] List with a tuple of min/max limits for each axis. If None, [(0, 1), (0, 1)] limits will be used.

n_grid_points [int, optional] Number of grid points. Default 30.

cmap [str or list or *matplotlib.pyplot.cm* or None, optional] Colormap to use. Could be a list of colors. If None and the number of dataset classes is <= *6*, colors will be chosen from ['blue', 'red', 'lightgreen', 'black', 'gray', 'cyan']. Otherwise the 'jet' colormap will be used.

class secml.figure._plots.c_plot_constraint.**CPlotConstraint**(*sp*, *default_params*)

Plot constraint on bi-dimensional feature spaces.

See also:

CPlot basic subplot functions.

CFigure creates and handle figures.

Attributes

class_type Defines class type.

logger Logger for current object.

n_lines Returns the number of lines inside current subplot.

verbose Verbosity level of logger output.

Methods

apply_params_fun(self)	Apply defined parameters to active subplot.
bar(self, left, height[, width, bottom])	Bar plot.
barh(self, bottom, width[, height, left])	Horizontal bar plot.
boxplot(self, x[, notch, sym, vert, whis, ...])	Make a box and whisker plot.
clabel(self, contour, *args, **kwargs)	Label a contour plot.
colorbar(self, mappable[, ticks])	Add colorbar to plot.
contour(self, x, y, z, *args, **kwargs)	Draw contour lines of a function.
contourf(self, x, y, z, *args, **kwargs)	Draw filled contour of a function.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
errorbar(self, x, y[, xerr, yerr])	Plot with error deltas in yerr and xerr.
fill_between(self, x, y1[, y2, where, ...])	Fill the area between two horizontal curves.
get_class_from_type(class_type)	Return the class associated with input type.
get_legend(self)	Returns the handler of current subplot legend.
get_legend_handles_labels(self)	Return handles and labels for legend contained by the subplot.
get_lines(self)	Return a list of lines contained by the subplot.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.

Continued on next page

Table 157 – continued from previous page

get_xticks_idx(self, xticks)	Returns the position of markers to plot.
grid(self[, grid_on, axis])	Draw grid for current plot.
hist(self, x, *args, **kwargs)	Plot a histogram.
imshow(self, img, *args, **kwargs)	Plot image.
legend(self, *args, **kwargs)	Create legend for plot.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
loglog(self, x[, y])	Plot with log scaling on both the x and y axis.
matshow(self, array, *args, **kwargs)	Plot an array as a matrix.
merge(self, sp)	Merge input subplot to active subplot.
plot(self, x[, y])	Plot a line.
plot_constraint(self, constraint[, ...])	Plot constraint bound.
plot_fgrads(self, gradf[, n_grid_points, ...])	Plot function gradient directions.
plot_fun(self, func[, multipoint, ...])	Plot a function (used for decision functions or boundaries).
plot_path(self, path[, path_style, ...])	Plot a path traversed by a point.
quiver(self, U, V[, X, Y, color, linestyle, ...])	A quiver plot displays velocity vectors as arrows with components (u,v) at the points (x,y).
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
scatter(self, x, y[, s, c])	Scatter plot of x vs y.
semilogx(self, x[, y])	Plot with log scaling on the x axis.
semilogy(self, x[, y])	Plot with log scaling on the y axis.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_axisbelow(self[, axisbelow])	Set axis ticks and gridlines below most artists.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
text(self, *args, **kwargs)	Create a Text instance at x, y with string text.
tick_params(self, *args, **kwargs)	Change the appearance of ticks and tick labels.
timed([msg])	Timer decorator.
title(self, text, *args, **kwargs)	Set a title for subplot.
xlabel(self, label, *args, **kwargs)	Set a label for the x axis.
xlim(self[, bottom, top])	Set axes x limits.
xscale(self, scale_type[, nonposx, basex])	Set scale for x axis.
xticklabels(self, labels, *args, **kwargs)	Set the xtick labels.
xticks(self, location_array, *args, **kwargs)	Set the x-tick locations and labels.
ylabel(self, label, *args, **kwargs)	Set a label for the y axis
ylim(self[, bottom, top])	Set axes y limits.
yscale(self, scale_type[, nonposy, basey])	Set scale for y axis.
yticklabels(self, labels, *args, **kwargs)	Set the ytick labels.
yticks(self, location_array, *args, **kwargs)	Set the y-tick locations and labels.

plot_constraint (*self*, *constraint*, *grid_limits=None*, *n_grid_points=30*)

Plot constraint bound.

Parameters

constraint [CConstraint] Constraint to be plotted.

grid_limits [list of tuple] List with a tuple of min/max limits for each axis. If None, [(0, 1),

(0, 1)] limits will be used.

n_grid_points [int, optional] Number of grid points. Default 30.

class secml.figure._plots.c_plot_ds.**CPlotDataset**(*sp*, *default_params*)

Plots a Dataset.

Custom plotting parameters can be specified.

Currently parameters default:

- show_legend: True
- grid: True

See also:

CDataset store and manage a dataset.

CPlot basic subplot functions.

CFigure creates and handle figures.

Attributes

class_type Defines class type.

logger Logger for current object.

n_lines Returns the number of lines inside current subplot.

verbose Verbosity level of logger output.

Methods

apply_params_ds(self)	Apply defined parameters to active subplot.
bar(self, left, height[, width, bottom])	Bar plot.
barh(self, bottom, width[, height, left])	Horizontal bar plot.
boxplot(self, x[, notch, sym, vert, whis, ...])	Make a box and whisker plot.
clabel(self, contour, *args, **kwargs)	Label a contour plot.
colorbar(self, mappable[, ticks])	Add colorbar to plot.
contour(self, x, y, z, *args, **kwargs)	Draw contour lines of a function.
contourf(self, x, y, z, *args, **kwargs)	Draw filled contour of a function.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
errorbar(self, x, y[, xerr, yerr])	Plot with error deltas in yerr and xerr.
fill_between(self, x, y1[, y2, where, ...])	Fill the area between two horizontal curves.
get_class_from_type(class_type)	Return the class associated with input type.
get_legend(self)	Returns the handler of current subplot legend.
get_legend_handles_labels(self)	Return handles and labels for legend contained by the subplot.
get_lines(self)	Return a list of lines contained by the subplot.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.

Continued on next page

Table 158 – continued from previous page

get_xticks_idx(self, xticks)	Returns the position of markers to plot.
grid(self[, grid_on, axis])	Draw grid for current plot.
hist(self, x, *args, **kwargs)	Plot a histogram.
imshow(self, img, *args, **kwargs)	Plot image.
legend(self, *args, **kwargs)	Create legend for plot.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
loglog(self, x[, y])	Plot with log scaling on both the x and y axis.
matshow(self, array, *args, **kwargs)	Plot an array as a matrix.
merge(self, sp)	Merge input subplot to active subplot.
plot(self, x[, y])	Plot a line.
plot_ds(self, dataset[, colors, markers])	Plot patterns of each class with a different color/marker.
plot_path(self, path[, path_style, ...])	Plot a path traversed by a point.
quiver(self, U, V[, X, Y, color, linestyle, ...])	A quiver plot displays velocity vectors as arrows with components (u,v) at the points (x,y).
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
scatter(self, x, y[, s, c])	Scatter plot of x vs y.
semilogx(self, x[, y])	Plot with log scaling on the x axis.
semilogy(self, x[, y])	Plot with log scaling on the y axis.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_axisbelow(self[, axisbelow])	Set axis ticks and gridlines below most artists.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
text(self, *args, **kwargs)	Create a Text instance at x, y with string text.
tick_params(self, *args, **kwargs)	Change the appearance of ticks and tick labels.
timed([msg])	Timer decorator.
title(self, text, *args, **kwargs)	Set a title for subplot.
xlabel(self, label, *args, **kwargs)	Set a label for the x axis.
xlim(self[, bottom, top])	Set axes x limits.
xscale(self, scale_type[, nonposx, basex])	Set scale for x axis.
xticklabels(self, labels, *args, **kwargs)	Set the xtick labels.
xticks(self, location_array, *args, **kwargs)	Set the x-tick locations and labels.
ylabel(self, label, *args, **kwargs)	Set a label for the y axis
ylim(self[, bottom, top])	Set axes y limits.
yscale(self, scale_type[, nonposy, basey])	Set scale for y axis.
yticklabels(self, labels, *args, **kwargs)	Set the ytick labels.
yticks(self, location_array, *args, **kwargs)	Set the y-tick locations and labels.

apply_params_ds (*self*)
Apply defined parameters to active subplot.

plot_ds (*self*, *dataset*, *colors=None*, *markers='o'*, **args*, ***kwargs*)
Plot patterns of each class with a different color/marker.

Parameters

dataset [CDataset] Dataset that contain samples which we want plot.

colors [list or None, optional] Color to be used for plotting each class. If a list, each color

will be assigned to a dataset's class, with repetitions if necessary. If None and the number of classes is 1, blue will be used. If None and the number of classes is 2, blue and red will be used. If None and the number of classes is > 2, 'jet' colormap is used.

markers [list or str, optional] Marker to use for plotting. Default is 'o' (circle). If a string, the same specified marker will be used for each class. If a list, must specify one marker for each dataset's class.

args, kwargs [any] Any optional argument for plots. If the number of classes is 2, a *plot* will be created. If the number of classes is > 2, a *scatter* plot will be created.

class secml.figure._plots.c_plot_fun.**CPlotFunction**(*sp*, *default_params*)

Plots a Function.

Custom plotting parameters can be specified.

Currently parameters default:

- show_legend: True
- grid: True

See also:

CPlot basic subplot functions.

CFigure creates and handle figures.

Attributes

class_type Defines class type.

logger Logger for current object.

n_lines Returns the number of lines inside current subplot.

verbose Verbosity level of logger output.

Methods

apply_params_fun(self)	Apply defined parameters to active subplot.
bar(self, left, height[, width, bottom])	Bar plot.
barh(self, bottom, width[, height, left])	Horizontal bar plot.
boxplot(self, x[, notch, sym, vert, whis, ...])	Make a box and whisker plot.
clabel(self, contour, *args, **kwargs)	Label a contour plot.
colorbar(self, mappable[, ticks])	Add colorbar to plot.
contour(self, x, y, z, *args, **kwargs)	Draw contour lines of a function.
contourf(self, x, y, z, *args, **kwargs)	Draw filled contour of a function.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
errorbar(self, x, y[, xerr, yerr])	Plot with error deltas in yerr and xerr.
fill_between(self, x, y1[, y2, where, ...])	Fill the area between two horizontal curves.
get_class_from_type(class_type)	Return the class associated with input type.
get_legend(self)	Returns the handler of current subplot legend.

Continued on next page

Table 159 – continued from previous page

get_legend_handles_labels(self)	Return handles and labels for legend contained by the subplot.
get_lines(self)	Return a list of lines contained by the subplot.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
get_xticks_idx(self, xticks)	Returns the position of markers to plot.
grid(self[, grid_on, axis])	Draw grid for current plot.
hist(self, x, *args, **kwargs)	Plot a histogram.
imshow(self, img, *args, **kwargs)	Plot image.
legend(self, *args, **kwargs)	Create legend for plot.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
loglog(self, x[, y])	Plot with log scaling on both the x and y axis.
matshow(self, array, *args, **kwargs)	Plot an array as a matrix.
merge(self, sp)	Merge input subplot to active subplot.
plot(self, x[, y])	Plot a line.
plot_fgrads(self, gradf[, n_grid_points, ...])	Plot function gradient directions.
plot_fun(self, func[, multipoint, ...])	Plot a function (used for decision functions or boundaries).
plot_path(self, path[, path_style, ...])	Plot a path traversed by a point.
quiver(self, U, V[, X, Y, color, linestyle, ...])	A quiver plot displays velocity vectors as arrows with components (u,v) at the points (x,y).
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
scatter(self, x, y[, s, c])	Scatter plot of x vs y.
semilogx(self, x[, y])	Plot with log scaling on the x axis.
semilogy(self, x[, y])	Plot with log scaling on the y axis.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_axisbelow(self[, axisbelow])	Set axis ticks and gridlines below most artists.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
text(self, *args, **kwargs)	Create a Text instance at x, y with string text.
tick_params(self, *args, **kwargs)	Change the appearance of ticks and tick labels.
timed([msg])	Timer decorator.
title(self, text, *args, **kwargs)	Set a title for subplot.
xlabel(self, label, *args, **kwargs)	Set a label for the x axis.
xlim(self[, bottom, top])	Set axes x limits.
xscale(self, scale_type[, nonposx, basex])	Set scale for x axis.
xticklabels(self, labels, *args, **kwargs)	Set the xtick labels.
xticks(self, location_array, *args, **kwargs)	Set the x-tick locations and labels.
ylabel(self, label, *args, **kwargs)	Set a label for the y axis
ylim(self[, bottom, top])	Set axes y limits.
yscale(self, scale_type[, nonposy, basey])	Set scale for y axis.
yticklabels(self, labels, *args, **kwargs)	Set the ytick labels.
yticks(self, location_array, *args, **kwargs)	Set the y-tick locations and labels.

apply_params_fun(*self*)

Apply defined parameters to active subplot.

plot_fgrads(*self*, *gradf*, *n_grid_points=30*, *grid_limits=None*, *color='k'*, *linestyle='-'*, *linewidth=1.0*, *alpha=1.0*, *func_args=()*, ***func_kwargs*)

Plot function gradient directions.

Parameters

gradf [function] Function that computes gradient directions.

n_grid_points [int] Number of grid points.

grid_limits [list of tuple] List with a tuple of min/max limits for each axis. If None, [(0, 1), (0, 1)] limits will be used.

color : Color of the gradient directions.

linestyle [str] ['solid' | 'dashed', 'dashdot', 'dotted' | (offset, on-off-dash-seq) | '-' | '–' | '-.' | ':' | 'None' | ' ' | '']

linewidth [float] Width of the line.

alpha [float] Transparency factor of the directions.

func_args, func_kwargs [any] Other arguments or keyword arguments to pass to *gradf*.

plot_fun(*self*, *func*, *multipoint=False*, *plot_background=True*, *plot_levels=True*, *levels=None*, *levels_color='k'*, *levels_style=None*, *levels_linewidth=1.0*, *n_colors=50*, *cmap='jet'*, *alpha=1.0*, *alpha_levels=1.0*, *vmin=None*, *vmax=None*, *colorbar=True*, *n_grid_points=30*, *grid_limits=None*, *func_args=()*, ***func_kwargs*)

Plot a function (used for decision functions or boundaries).

Parameters

func [unbound function] Function to be plotted.

multipoint [bool, optional] If True, all grid points will be passed to the function. If False (default), function is iterated over each point of the grid.

plot_background [bool, optional] Specifies whether to plot the value of func at each point in the background using a colorbar.

plot_levels [bool, optional] Specify if function levels should be plotted (default True).

levels [list or None, optional] List of levels to be plotted. If None, 0 (zero) level will be plotted.

levels_color [str or tuple or None, optional] If None, the colormap specified by cmap will be used. If a string, like 'k', all levels will be plotted in this color. If a tuple of colors (string, float, rgb, etc), different levels will be plotted in different colors in the order specified. Default 'k'.

levels_style [[None | 'solid' | 'dashed' | 'dashdot' | 'dotted']] If levels_style is None, the default is 'solid'. levels_style can also be an iterable of the above strings specifying a set of levels_style to be used. If this iterable is shorter than the number of contour levels it will be repeated as necessary.

levels_linewidth [float or list of floats, optional] The line width of the contour lines. Default 1.0.

n_colors [int, optional] Number of color levels of background plot. Default 50.

cmap [str or list or *matplotlib.pyplot.cm*, optional] Colormap to use (default 'jet'). Could be a list of colors.

alpha [float, optional] The alpha blending value of the background. Default 1.0.

alpha_levels [float, optional] The alpha blending value of the levels. Default 1.0.

vmin, vmax [float or None, optional] Limits of the colors used for function plotting. If None, colors are determined by the colormap.

colorbar [bool, optional] True if colorbar should be displayed.

n_grid_points [int, optional] Number of grid points.

grid_limits [list of tuple, optional] List with a tuple of min/max limits for each axis. If None, [(0, 1), (0, 1)] limits will be used.

func_args, func_kwargs Other arguments or keyword arguments to pass to *func*.

Examples

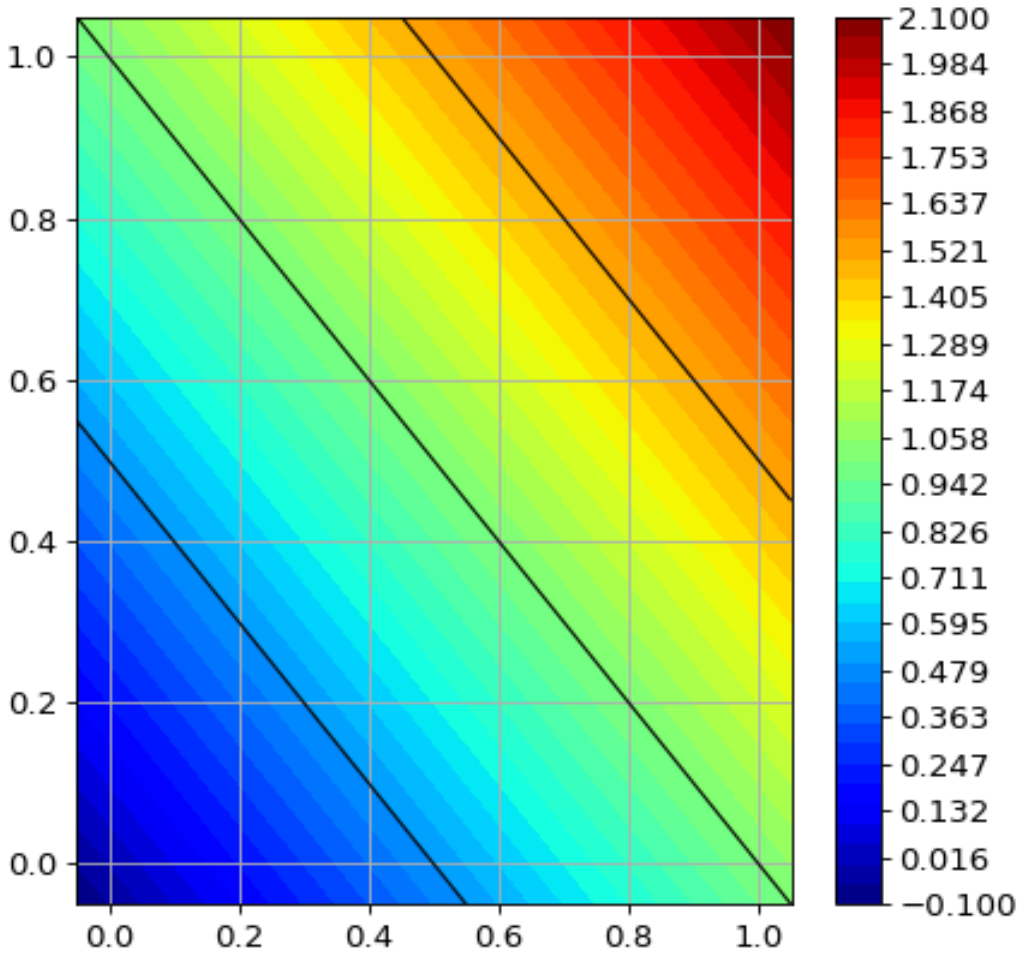
```
from secml.array import CArray
from secml.figure import CFigure


# we must define a function that take an array and return a value for every␣
↪row
def f(array):
    res = CArray.zeros(array.shape[0])
    for r in range(array.shape[0]):
        x = array[r, 0]
        y = array[r, 1]
        res[r] = x + y
    return res


fig = CFigure()

fig.sp.plot_fun(f, levels=[.5, 1, 1.5])

fig.sp.grid()
fig.show()
```

class secml.figure._plots.c_plot_metric.**CPlotMetric**(*sp*, *default_params*)

Plots of performance evaluation metrics.

Currently parameters default for ROC plots:

- show_legend: True
- ylabel: 'True Positive Rate (%)'
- xlabel: 'False Positive Rate (%)'
- yticks: [0, 20, 40, 60, 80, 100]
- yticklabels: see yticks
- xticks: list. [0.1, 0.5, 1, 2, 5, 10, 20, 50, 100]
- xticklabels: see xticks
- ylim: (0.1, 100)
- xlim: (0, 100)
- grid: True


1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
2.100
1.984
1.868
1.753
1.637
1.521
1.405
1.289
1.174
1.058
0.942
0.826
0.711
0.595
0.479
0.363
0.247
0.132
0.016
−0.100

See also:

CPlot basic subplot functions.

CFigure creates and handle figures.

Attributes

class_type Defines class type.

logger Logger for current object.

n_lines Returns the number of lines inside current subplot.

verbose Verbosity level of logger output.

Methods

apply_params_roc(self)	Apply defined parameters to active subplot.
bar(self, left, height[, width, bottom])	Bar plot.
barh(self, bottom, width[, height, left])	Horizontal bar plot.
boxplot(self, x[, notch, sym, vert, whis, ...])	Make a box and whisker plot.
clabel(self, contour, *args, **kwargs)	Label a contour plot.
colorbar(self, mappable[, ticks])	Add colorbar to plot.
contour(self, x, y, z, *args, **kwargs)	Draw contour lines of a function.
contourf(self, x, y, z, *args, **kwargs)	Draw filled contour of a function.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
errorbar(self, x, y[, xerr, yerr])	Plot with error deltas in yerr and xerr.
fill_between(self, x, y1[, y2, where, ...])	Fill the area between two horizontal curves.
get_class_from_type(class_type)	Return the class associated with input type.
get_legend(self)	Returns the handler of current subplot legend.
get_legend_handles_labels(self)	Return handles and labels for legend contained by the subplot.
get_lines(self)	Return a list of lines contained by the subplot.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
get_xticks_idx(self, xticks)	Returns the position of markers to plot.
grid(self[, grid_on, axis])	Draw grid for current plot.
hist(self, x, *args, **kwargs)	Plot a histogram.
imshow(self, img, *args, **kwargs)	Plot image.
legend(self, *args, **kwargs)	Create legend for plot.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
loglog(self, x[, y])	Plot with log scaling on both the x and y axis.
matshow(self, array, *args, **kwargs)	Plot an array as a matrix.
merge(self, sp)	Merge input subplot to active subplot.
plot(self, x[, y])	Plot a line.

Continued on next page

Table 160 – continued from previous page

plot_confusion_matrix(self, y_true, y_pred)	Plot a confusion matrix.
plot_path(self, path[, path_style, ...])	Plot a path traversed by a point.
plot_roc(self, fpr, tpr[, label, style, logx])	Plot a ROC curve given input fpr and tpr.
plot_roc_mean(self, roc[, label, ...])	Plot the mean of ROC curves.
plot_roc_reps(self, roc[, label, ...])	Plot all input ROC curves.
quiver(self, U, V[, X, Y, color, linestyle, ...])	A quiver plot displays velocity vectors as arrows with components (u,v) at the points (x,y).
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
scatter(self, x, y[, s, c])	Scatter plot of x vs y.
semilogx(self, x[, y])	Plot with log scaling on the x axis.
semilogy(self, x[, y])	Plot with log scaling on the y axis.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_axisbelow(self[, axisbelow])	Set axis ticks and gridlines below most artists.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
text(self, *args, **kwargs)	Create a Text instance at x, y with string text.
tick_params(self, *args, **kwargs)	Change the appearance of ticks and tick labels.
timed([msg])	Timer decorator.
title(self, text, *args, **kwargs)	Set a title for subplot.
xlabel(self, label, *args, **kwargs)	Set a label for the x axis.
xlim(self[, bottom, top])	Set axes x limits.
xscale(self, scale_type[, nonposx, basex])	Set scale for x axis.
xticklabels(self, labels, *args, **kwargs)	Set the xtick labels.
xticks(self, location_array, *args, **kwargs)	Set the x-tick locations and labels.
ylabel(self, label, *args, **kwargs)	Set a label for the y axis
ylim(self[, bottom, top])	Set axes y limits.
yscale(self, scale_type[, nonposy, basey])	Set scale for y axis.
yticklabels(self, labels, *args, **kwargs)	Set the ytick labels.
yticks(self, location_array, *args, **kwargs)	Set the y-tick locations and labels.

apply_params_roc(*self*)
Apply defined parameters to active subplot.

plot_confusion_matrix(*self*, *y_true*, *y_pred*, *normalize=False*, *labels=None*, *title=None*, *cmap='Blues'*, *colorbar=False*)
Plot a confusion matrix.

y_true [CArray] True labels.

y_pred [CArray] Predicted labels.

normalize [bool, optional] If True, normalize the confusion matrix in 0/1. Default False.

labels [list, optional] List with the label of each class.

title: str, optional Title of the plot. Default None.

cmap: str or matplotlib.pyplot.cm, optional Colormap to use for plotting. Default 'Blues'.

colorbar [bool, optional] If True, show the colorbar side of the matrix. Default False.

plot_roc(*self*, *fpr*, *tpr*, *label=None*, *style=None*, *logx=True*)
Plot a ROC curve given input fpr and tpr.

Curves will be plotted inside the active figure or a new figure will be created using default parameters.

Parameters

fpr [CArray] Array with False Positive Rats.

tpr [CArray] Array with False Positive Rates.

label [str or None, optional] Label to assign to the roc.

style [str or None, optional] Style of the roc plot.

logx [bool, optional] If True (default), logarithmic scale will be used for fpr axis.

Returns

roc_plot [CFigure] Figure after this plot session.

plot_roc_mean(*self*, *roc*, *label=None*, *invert_tpr=False*, *style=None*, *plot_std=False*, *logx=True*)
Plot the mean of ROC curves.

Curves will be plotted inside the active figure or a new figure will be created using default parameters.

Parameters

roc [CRoc] Roc curves to plot.

label [str or None, optional] Label to assign to the roc.

invert_tpr [bool] True if 1 - tpr (False Negative Rates) should be plotted on y axis. Default False.

style [str or None, optional] Style of the roc plot. If a string, must follow the matplotlib convention: '[color][marker][line]'.

plot_std [bool (default False)] If True, standard deviation of True Positive Rates will be plotted.

logx [bool, optional] If True (default), logarithmic scale will be used for fpr axis.

Returns

roc_plot [CFigure] Figure after this plot session.

plot_roc_reps(*self*, *roc*, *label=None*, *invert_tpr=False*, *logx=True*)
Plot all input ROC curves.

Curves will be plotted inside the active figure or a new figure will be created using default parameters.

Parameters

roc [CRoc] Roc curves to plot.

label [str or None, optional] Label to assign to the roc. Repetition number will be appended using the following convention:

- If label is None -> "rep 'i'"
- If label is not None -> "*label* (rep *i*)"

invert_tpr [bool] True if 1 - tpr (False Negative Rates) should be plotted on y axis. Default False.

logx [bool, optional] If True (default), logarithmic scale will be used for fpr axis.

Returns

roc_plot [CFigure] Figure after this plot session.

class secml.figure._plots.c_plot_sec_eval.**CPlotSecEval**(*sp*, *default_params*)

Plots Classifier Security Evaluation results.

This class creates a figure plotting in a fancy and standard style data from .*CSecEvalData* class.

Custom plotting parameters can be specified.

Currently parameters default:

- *show_legend*: True. Set False to hide *show_legend* on next plot.
- grid: True.

See also:

CPlot basic subplot functions.

CFigure creates and handle figures.

Attributes

class_type Defines class type.

logger Logger for current object.

n_lines Returns the number of lines inside current subplot.

verbose Verbosity level of logger output.

Methods

apply_params_sec_eval(self)	Apply defined parameters to active subplot.
bar(self, left, height[, width, bottom])	Bar plot.
barh(self, bottom, width[, height, left])	Horizontal bar plot.
boxplot(self, x[, notch, sym, vert, whis, ...])	Make a box and whisker plot.
clabel(self, contour, *args, **kwargs)	Label a contour plot.
colorbar(self, mappable[, ticks])	Add colorbar to plot.
contour(self, x, y, z, *args, **kwargs)	Draw contour lines of a function.
contourf(self, x, y, z, *args, **kwargs)	Draw filled contour of a function.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
errorbar(self, x, y[, xerr, yerr])	Plot with error deltas in yerr and xerr.
fill_between(self, x, y1[, y2, where, ...])	Fill the area between two horizontal curves.
get_class_from_type(class_type)	Return the class associated with input type.
get_legend(self)	Returns the handler of current subplot legend.
get_legend_handles_labels(self)	Return handles and labels for legend contained by the subplot.
get_lines(self)	Return a list of lines contained by the subplot.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
get_xticks_idx(self, xticks)	Returns the position of markers to plot.
grid(self[, grid_on, axis])	Draw grid for current plot.
hist(self, x, *args, **kwargs)	Plot a histogram.

Continued on next page

Table 161 – continued from previous page

imshow(self, img, *args, **kwargs)	Plot image.
legend(self, *args, **kwargs)	Create legend for plot.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
loglog(self, x[, y])	Plot with log scaling on both the x and y axis.
matshow(self, array, *args, **kwargs)	Plot an array as a matrix.
merge(self, sp)	Merge input subplot to active subplot.
plot(self, x[, y])	Plot a line.
plot_path(self, path[, path_style, ...])	Plot a path traversed by a point.
plot_sec_eval(self, sec_eval_data[, metric, ...])	Plot the Security Evaluation Curve using desired metric.
quiver(self, U, V[, X, Y, color, linestyle, ...])	A quiver plot displays velocity vectors as arrows with components (u,v) at the points (x,y).
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
scatter(self, x, y[, s, c])	Scatter plot of x vs y.
semilogx(self, x[, y])	Plot with log scaling on the x axis.
semilogy(self, x[, y])	Plot with log scaling on the y axis.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_axisbelow(self[, axisbelow])	Set axis ticks and gridlines below most artists.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
text(self, *args, **kwargs)	Create a Text instance at x, y with string text.
tick_params(self, *args, **kwargs)	Change the appearance of ticks and tick labels.
timed([msg])	Timer decorator.
title(self, text, *args, **kwargs)	Set a title for subplot.
xlabel(self, label, *args, **kwargs)	Set a label for the x axis.
xlim(self[, bottom, top])	Set axes x limits.
xscale(self, scale_type[, nonposx, basex])	Set scale for x axis.
xticklabels(self, labels, *args, **kwargs)	Set the xtick labels.
xticks(self, location_array, *args, **kwargs)	Set the x-tick locations and labels.
ylabel(self, label, *args, **kwargs)	Set a label for the y axis
ylim(self[, bottom, top])	Set axes y limits.
yscale(self, scale_type[, nonposy, basey])	Set scale for y axis.
yticklabels(self, labels, *args, **kwargs)	Set the ytick labels.
yticks(self, location_array, *args, **kwargs)	Set the y-tick locations and labels.

apply_params_sec_eval(*self*)
Apply defined parameters to active subplot.

plot_sec_eval(*self*, *sec_eval_data*, *metric='accuracy'*, *mean=False*, *percentage=False*, *show_average=False*, *label=None*, *linestyle='-'*, *color=None*, *marker=None*, *metric_args=()*)
Plot the Security Evaluation Curve using desired metric.

Parameters

sec_eval_data [CSecEvalData or list] A single CSecEvalData object or a list with multiple repetitions.

metric [str or CMetric, optional] Metric to be evaluated. Default 'accuracy'.

mean [bool, optional] If True, the mean of all sec eval repetitions will be computed. Default False..

percentage [bool, optional] If True, values will be displayed in percentage. Default False.

show_average [bool, optional] If True, the average along the sec eval parameters will be shown in legend. Default False.

label [str, optional] Label of the sec eval curve. Default None.

linestyle [str, optional] Style of the curve. Default '-'.

color [str or None, optional] Color of the curve. If None (default) the plot engine will chose.

marker [str or None, optional] Style of the markers. Default None.

metric_args Any other argument for the metric.

class secml.figure._plots.c_plot_stats.**CPlotStats**(*sp*, *default_params*)

Plots for statistical functions.

Custom plotting parameters can be specified.

Currently parameters default:

- *show_legend*: True.
- grid: True.

See also:

CPlot basic subplot functions.

CFigure creates and handle figures.

Attributes

class_type Defines class type.

logger Logger for current object.

n_lines Returns the number of lines inside current subplot.

verbose Verbosity level of logger output.

Methods

apply_params_stats(self)	Apply defined parameters to active subplot.
bar(self, left, height[, width, bottom])	Bar plot.
barh(self, bottom, width[, height, left])	Horizontal bar plot.
boxplot(self, x[, notch, sym, vert, whis, ...])	Make a box and whisker plot.
clabel(self, contour, *args, **kwargs)	Label a contour plot.
colorbar(self, mappable[, ticks])	Add colorbar to plot.
contour(self, x, y, z, *args, **kwargs)	Draw contour lines of a function.
contourf(self, x, y, z, *args, **kwargs)	Draw filled contour of a function.
copy(self)	Returns a shallow copy of current class.
create([class_item])	This method creates an instance of a class with given type.
deepcopy(self)	Returns a deep copy of current class.
errorbar(self, x, y[, xerr, yerr])	Plot with error deltas in yerr and xerr.

Continued on next page

Table 162 – continued from previous page

fill_between(self, x, y1[, y2, where, ...])	Fill the area between two horizontal curves.
get_class_from_type(class_type)	Return the class associated with input type.
get_legend(self)	Returns the handler of current subplot legend.
get_legend_handles_labels(self)	Return handles and labels for legend contained by the subplot.
get_lines(self)	Return a list of lines contained by the subplot.
get_params(self)	Returns the dictionary of class hyperparameters.
get_state(self)	Returns the object state dictionary.
get_subclasses()	Get all the subclasses of the calling class.
get_xticks_idx(self, xticks)	Returns the position of markers to plot.
grid(self[, grid_on, axis])	Draw grid for current plot.
hist(self, x, *args, **kwargs)	Plot a histogram.
imshow(self, img, *args, **kwargs)	Plot image.
legend(self, *args, **kwargs)	Create legend for plot.
list_class_types()	This method lists all types of available subclasses of calling one.
load(path)	Loads object from file.
load_state(self, path)	Sets the object state from file.
loglog(self, x[, y])	Plot with log scaling on both the x and y axis.
matshow(self, array, *args, **kwargs)	Plot an array as a matrix.
merge(self, sp)	Merge input subplot to active subplot.
plot(self, x[, y])	Plot a line.
plot_path(self, path[, path_style, ...])	Plot a path traversed by a point.
plot_prob_density(self, scores, ts, **params)	Plot density estimation of benign and malicious class.
quiver(self, U, V[, X, Y, color, linestyle, ...])	A quiver plot displays velocity vectors as arrows with components (u,v) at the points (x,y).
save(self, path)	Save class object to file.
save_state(self, path)	Store the object state to file.
scatter(self, x, y[, s, c])	Scatter plot of x vs y.
semilogx(self, x[, y])	Plot with log scaling on the x axis.
semilogy(self, x[, y])	Plot with log scaling on the y axis.
set(self, param_name, param_value[, copy])	Set a parameter of the class.
set_axisbelow(self[, axisbelow])	Set axis ticks and gridlines below most artists.
set_params(self, params_dict[, copy])	Set all parameters passed as a dictionary {key: value}.
set_state(self, state_dict[, copy])	Sets the object state using input dictionary.
text(self, *args, **kwargs)	Create a Text instance at x, y with string text.
tick_params(self, *args, **kwargs)	Change the appearance of ticks and tick labels.
timed([msg])	Timer decorator.
title(self, text, *args, **kwargs)	Set a title for subplot.
xlabel(self, label, *args, **kwargs)	Set a label for the x axis.
xlim(self[, bottom, top])	Set axes x limits.
xscale(self, scale_type[, nonposx, basex])	Set scale for x axis.
xticklabels(self, labels, *args, **kwargs)	Set the xtick labels.
xticks(self, location_array, *args, **kwargs)	Set the x-tick locations and labels.
ylabel(self, label, *args, **kwargs)	Set a label for the y axis
ylim(self[, bottom, top])	Set axes y limits.
yscale(self, scale_type[, nonposy, basey])	Set scale for y axis.
yticklabels(self, labels, *args, **kwargs)	Set the ytick labels.

Continued on next page

Table 162 – continued from previous page

yticks(self, location_array, *args, **kwargs)	Set the y-tick locations and labels.

apply_params_stats(*self*)
Apply defined parameters to active subplot.

plot_prob_density(*self*, *scores*, *ts*, ***params*)
Plot density estimation of benign and malicious class.

secml.figure._plots.plot_utils.**create_points_grid**(*grid_limits*, *n_grid_points*)
Creates a grid of points.

Parameters

grid_limits [list of tuple] List with a tuple of min/max limits for each axis. If None, [(0, 1), (0, 1)] limits will be used.

n_grid_points [int] Number of grid points.

4.7.13 secml.parallel

parfor

secml.parallel.parfor.**parfor**(*task*, *processes*, *args*)
Parallel For.

Applies a function *task* to each argument in *args*, using a pool of concurrent processes.

Parameters

task [function] Function object that should process each element in args.

processes [int] Maximum number of concurrent processes to be used in the pool. If higher than multiprocessing.cpu_count(), all processor's cores will be used.

args [any] Iterable object, where each element is an argument for task.

Returns

out [iterable] Iterable object containing the output of task(arg) for each arg in args.

secml.parallel.parfor.**parfor2**(*task*, *n_reps*, *processes*, **args*)
Parallel For.

Run function *task* using each argument in *args* as input, using a pool of concurrent processes. The *task* should take as first input the index of parfor iteration.

Parameters

task [function] Function object that should process each element in *args*.

n_reps [int] Number of times the *task* should be run.

processes [int] Maximum number of concurrent processes to be used in the pool. If higher than *multiprocessing.cpu_count()*, all processor's cores will be used.

args [any, optional] Tuple with input arguments for *task*.

Returns

out [list] List with iteration output, sorted (rep1, rep2, . . . , repN).

4.7.14 secml.utils

CLog

class secml.utils.c_log.**CLog**(*level=None*, *logger_id=None*, *add_stream=True*, *file_handler=None*, *propagate=False*)

Bases: object

Manager for logging and logfiles.

Logger can be used to save important runtime code information to disk instead of built-in function 'print'. Along with any print-like formatted string, the logger stores full time stamp and calling class name.

The default filename of a log file is `logs.log`. This will be placed in the same directory of the calling file.

Logging levels currently available and target purpose:

- DISABLE - 100: disable all logging.
- CRITICAL - 50: critical error messages.
- ERROR - 40: standard error messages.
- WARNING - 30: standard error messages.
- INFO - 20: general info logging.
- DEBUG - 10: debug logging only.

Logger is fully integrated to the *CTimer* class in order to log performance of a desired method or routine.

Parameters

level [LOG_LEVEL, int or None, optional] Initial logging level. Default is None, meaning that the current logging level will be preserved if the logger has already been created.

logger_id [str or None, optional] Identifier of the logger. Default None. If None, creates a logger which is the root of the hierarchy

add_stream [bool, optional] If True, attach a stream handler to the logger. Default True. A stream handler prints to stdout the logged messages.

file_handler [str or None, optional] If a string, attach a file handler to the logger. Default None. A file handler stores to the specified path the logged messages.

propagate [bool, optional] If True, messages logged to this logger will be passed to the handlers of higher level (ancestor) loggers, in addition to any handler attached to this logger. Default False.

See also:

CTimer Manages performance monitoring and logging.

Notes

Unlike most of the Python logging modules, our implementation can be fully used inside parallelized code.

Examples

```
>>> from secml.array import CArray
>>> from secml.utils import CLog
```

```
>>> log = CLog().warning("{:}".format(CArray([1,2,3])))
... - WARNING - CArray([1 2 3])
```

Attributes

level Return logging level.

logger_id Return identifier of the logger.

propagate If True, events logged will be passed to the handlers of higher level (ancestor) loggers.

Methods

attach_file(self, filepath)	Adds a file handler to the logger.
attach_stream(self)	Adds a stream handler to the logger.
catch_warnings([record])	A context manager that copies and restores the warnings filter upon exiting the context.
critical(self, msg, *args, **kwargs)	Logs a message with level CRITICAl on this logger.
debug(self, msg, *args, **kwargs)	Logs a message with level DEBUG on this logger.
error(self, msg, *args, **kwargs)	Logs a message with level ERROR on this logger.
filterwarnings(action[, message, category, ...])	Insert an entry into the list of warnings filters (at the front).
get_child(self, name)	Return a child logger associated with ancestor.
info(self, msg, *args, **kwargs)	Logs a message with level INFO on this logger.
log(self, level, msg, *args, **kwargs)	Logs a message with specified level on this logger.
remove_handler_file(self, filepath)	Removes the file handler from the logger.
remove_handler_stream(self)	Removes the stream handler from the logger.
set_level(self, level)	Sets logging level of the logger.
timed(self[, msg])	Timer decorator.
timer(self[, msg])	Starts a timed codeblock.
warning(self, msg, *args, **kwargs)	Logs a message with level WARNING on this logger.

attach_file(*self*, *filepath*)
Adds a file handler to the logger.

attach_stream(*self*)
Adds a stream handler to the logger.

static catch_warnings(*record=False*)
A context manager that copies and restores the warnings filter upon exiting the context.

Wrapper of *warnings.catch_warnings*.

Parameters

record [bool, optional] If False (the default), the context manager returns None on entry. If True, a list is returned that is progressively populated with warning objects as seen by the context manager.

critical(*self*, *msg*, **args*, ***kwargs*)
Logs a message with level CRITICAl on this logger.

See *CLog.log* for details on args and kwargs.

debug(*self*, *msg*, **args*, ***kwargs*)
Logs a message with level DEBUG on this logger.

See *CLog.log* for details on args and kwargs.

error(*self*, *msg*, **args*, ***kwargs*)
Logs a message with level ERROR on this logger.

See *CLog.log* for details on args and kwargs.

static filterwarnings(*action*, *message=''*, *category=<class 'Warning'>*, *module=''*, *lineno=0*, *append=False*)
Insert an entry into the list of warnings filters (at the front).

Wrapper of *warnings.filterwarnings*.

Parameters

action [str] One of “error”, “ignore”, “always”, “default”, “module”, or “once”.

message [str, optional] A regex that the warning message must match.

category [class, optional] A class that the warning must be a subclass of. Default *Warning*.

module [str, optional] A regex that the module name must match.

lineno [int, optional] An integer line number, 0 (default) matches all warnings.

append [bool, optional] If true, append to the list of filters.

get_child(*self*, *name*)
Return a child logger associated with ancestor.

Parameters

name [str-like] Identifier of the child logger. Can be any object safely convertible to string (int, float, etc.)

Returns

child_logger [logger] Instance of the child logger.

info(*self*, *msg*, **args*, ***kwargs*)
Logs a message with level INFO on this logger.

See *CLog.log* for details on args and kwargs.

property level
Return logging level.

log(*self*, *level*, *msg*, **args*, ***kwargs*)
Logs a message with specified level on this logger.

The msg is the message format string, and the args are the arguments which are merged into msg using the string formatting operator.

There are two keyword arguments in kwargs which are inspected: exc_info which, if it does not evaluate as false, causes exception information to be added to the logging message. If an exception tuple (in the

format returned by sys.exc_info()) is provided, it is used; otherwise, sys.exc_info() is called to get the exception information.

The second keyword argument is extra which can be used to pass a dictionary which is used to populate the __dict__ of the LogRecord created for the logging event with user-defined attributes.

property logger_id
Return identifier of the logger.

property propagate
If True, events logged will be passed to the handlers of higher level (ancestor) loggers.

remove_handler_file(*self*, *filepath*)
Removes the file handler from the logger.

remove_handler_stream(*self*)
Removes the stream handler from the logger.

set_level(*self*, *level*)
Sets logging level of the logger.

timed(*self*, *msg=None*)
Timer decorator.

Returns a decorator that can be used to measure execution time of any method. Performance data will be stored inside the calling logger. Messages will be logged using the DEBUG logging level. As this decorator accepts optional arguments, must be called as a method. See examples.

Parameters

msg [str or None, optional] Custom message to display when entering the timed block. If None, "Entering timed block *function_name*..." will printed.

Examples

```
>>> from secml.array import CArray
>>> from secml.utils import CLog
```

```
>>> log = CLog()
>>> log.set_level(10)
>>> @log.timed()
... def abc():
...     print("Hello world!")
```

```
>>> abc()
Hello world!
```

timer(*self*, *msg=None*)
Starts a timed codeblock.

Returns an instance of context manager *CTimer*. Performance data will be stored inside the calling logger. Messages will be logged using the DEBUG logging level.

Parameters

msg [str or None, optional] Custom message to display when entering the timed block. If None, "Entering timed block..." will printed.

Examples

```
>>> from secml.array import CArray
>>> from secml.utils import CLog
```

```
>>> log = CLog()
>>> log.set_level(10)
>>> with log.timer("Timing the instruction..."):
...     a = CArray([1,2,3])
2... - root - DEBUG - Timing the instruction...
2... - root - DEBUG - Elapsed time: ... ms
```

warning(*self*, *msg*, **args*, ***kwargs*)

Logs a message with level WARNING on this logger.

See *CLog.log* for details on args and kwargs.

class secml.utils.c_log.**CTimer**(*log=None*, *msg=None*)

Bases: object

Context manager for performance logging

The code inside the specific context will be timed and performance data printed and/or logged.

This class fully integrates with *CLog* in order to store to disk performance data. When no logger is specified, data is printed on the console output.

Times are always stored in milliseconds (ms).

Parameters

log [CLog or None, optional] Instance of *CLog* class to be used as performance logger. If a logger is specified, timer data will not be printed on console.

msg [str or None, optional] Custom message to display when entering the timed block. If None, "Entering timed block *function_name*..." will printed.

See also:

CLog CLog and store runtime information on disk.

Examples

```
>>> from secml.array import CArray
>>> from secml.utils import CTimer
```

```
>>> with CTimer() as t:
...     a = CArray([1,2,3])
Entering timed block...
Elapsed time: ... ms
```

```
>>> with CTimer(msg="Timing the instruction...") as t:
...     a = CArray([1,2,3])
Timing the instruction...
Elapsed time: ... ms
```

```
>>> from secml.utils import CLog
>>> logger = CLog()
>>> logger.set_level(10)
>>> with CTimer(logger) as t:
...     a = CArray([1,2,3])
2... - root - DEBUG - Entering timed block...
2... - root - DEBUG - Elapsed time: ... ms
```

Attributes

step Return time elapsed from timer start (milliseconds).

Methods

timed([log, msg])	Timer decorator.

property step
Return time elapsed from timer start (milliseconds).

static timed(*log=None*, *msg=None*)
Timer decorator.

Returns a decorator that can be used to measure execution time of any method. As this decorator accepts optional arguments, must be called as a method. See examples.

Parameters

log [CLog or None, optional] Instance of *CLog* class to be used as performance logger. If a logger is specified, timer data will not be printed on console.

msg [str or None, optional] Custom message to display when entering the timed block. If None, "Entering timed block..." will printed.

See also:

CLog CLog and store runtime information on disk.

Examples

```
>>> from secml.array import CArray
>>> from secml.utils import CTimer
```

```
>>> @CTimer.timed()
... def abc():
...     print("Hello world!")
```

```
>>> abc()
Entering timed block `abc`...
Hello world!
Elapsed time: ... ms
```

c_file_manager

secml.utils.c_file_manager.**folder_exist**(*folder_path*)
Test whether a folder exists.

Returns False for broken symbolic links.

secml.utils.c_file_manager.**file_exist**(*file_path*)
Test whether a file exists.

Returns False for broken symbolic links.

secml.utils.c_file_manager.**make_folder_incwd**(*folder_name*, *mode=511*)
Create a directory named folder_name inside current working directory (cwd).

Parameters

folder_name [str] Desired name for the new folder.

mode [oct, optional] Octal literal representing the numeric mode to use. On some systems, mode is ignored. Where it is used, the current umask value is first masked out. If bits other than the last 9 (i.e. the last 3 digits of the octal representation of the mode) are set, their meaning is platform-dependent. On some platforms, they are ignored and you should call chmod() explicitly to set them. Default *0o777*.

See also:

make_folder Create a directory given full path.

secml.utils.c_file_manager.**make_folder**(*folder_path*, *mode=511*)
Create a directory inside folder_path with numeric mode 'mode'.

All intermediate-level directories needed to contain the leaf directory will be recursively made.

Parameters

folder_path [str] Desired path for the new folder.

mode [oct, optional] Octal literal representing the numeric mode to use. On some systems, mode is ignored. Where it is used, the current umask value is first masked out. If bits other than the last 9 (i.e. the last 3 digits of the octal representation of the mode) are set, their meaning is platform-dependent. On some platforms, they are ignored and you should call chmod() explicitly to set them. Default *0o777*.

See also:

make_folder_inpath Create a directory inside a specific folder.

secml.utils.c_file_manager.**remove_folder**(*folder_path*, *force=False*)
Remove (delete) the directory path.

Path must point to a directory (but not a symbolic link to a directory).

Parameters

folder_path [str] Absolute or relative path to folder to remove.

force [bool, optional] By default, if force is False, directory is removed only if empty, otherwise, OSError is raised. Set to True in order to remove the whole directory and its subdirectories.

secml.utils.c_file_manager.**make_rand_folder**(*folder_path=None*, *custom_name=None*)
Create a random named folder.

Random name will be selected inside the integer range between 1 and 1 million [1, 1kk).

Parameters

folder_path [str, optional] Path where to create the new directory. If None, folder will be created inside calling file folder.

custom_name [str, optional] Custom name to add before the random ID number. An underscore is placed between ID and custom_name.

Returns

target_path [str] Absolute path of created directory.

Notes

There is a small chance that randomly generated folder already exists. Just run the function again :)

secml.utils.c_file_manager.**abspath**(*file_name*)

Return the absolute path to file.

File name, as well as directory separator, is not added to the end of the returned path.

Examples

```
>>> import secml.utils.c_file_manager as fm
```

```
>>> cur_file = fm.split(__file__)[1]  # Getting only the filename
>>> cur_file
'c_folder_manager.py'
>>> fm.abspath(cur_file)[-12:]
'/secml/utils'
```

secml.utils.c_file_manager.**normpath**(*path*)

Normalize a pathname.

Normalize a pathname by collapsing redundant separators and up-level references so that A//B, A/B/, A/./B and A/foo/../B all become A/B. This string manipulation may change the meaning of a path that contains symbolic links. On Windows, it converts forward slashes to backward slashes.

Examples

```
>>> import secml.utils.c_file_manager as fm
```

```
>>> cur_path = fm.split(__file__)[0]  # Getting only the filename
>>> cur_path
'---/src/secml/utils'
>>> upper_path = fm.join(cur_path, '..', '..')
>>> upper_path
'---/src/secml/utils/../..'
>>> fm.normpath(upper_path)
'---/src'
```

secml.utils.c_file_manager.**join**(**paths*)

Join one or more path components intelligently.

The return value is the concatenation of path and any members of **paths* with exactly one directory separator (os.sep) following each non-empty part except the last, meaning that the result will only end in a separator if the last part is empty. If a component is an absolute path, all previous components are thrown away and joining continues from the absolute path component.

See also:

split Split the pathname path into a pair (head, tail).

secml.utils.c_file_manager.**split**(*path*)

Split the pathname path into a pair (head, tail).

Tail is the last pathname component and head is everything leading up to that. The tail part will never contain a slash; if path ends in a slash, tail will be empty. If there is no slash in path, head will be empty. If path is empty, both head and tail are empty. Trailing slashes are stripped from head unless it is the root (one or more slashes only). In all cases, join(head, tail) returns a path to the same location as path (but the strings may differ).

Returns

out_split [tuple of str] A tuple of strings consisting of (head, tail), where tail is the last pathname component and head is everything leading up to that.

See also:

join Join one or more path components intelligently.

Examples

```
>>> import secml.utils.c_file_manager as fm
```

```
>>> path = fm.join('dir1','dir2','dir3')
>>> path
'dir1/dir2/dir3'
>>> print(fm.split(path))
('dir1/dir2', 'dir3')
```

secml.utils.c_file_manager.**expanduser**(*path*)

Replace user path shortcut with real user path.

On Unix and Windows, return *path* with an initial ~ or ~user replaced by that user's home directory.

On Unix, an initial ~ is replaced by the environment variable HOME if it is set; otherwise the current user's home directory is looked up in the password directory through the built-in module pwd. An initial ~user is looked up directly in the password directory.

On Windows, HOME and USERPROFILE will be used if set, otherwise a combination of HOMEPATH and HOMEDRIVE will be used. An initial ~user is handled by stripping the last directory component from the created user path derived above.

If the expansion fails or if the path does not begin with a tilde, the path is returned unchanged.

Examples

```
>>> import secml.utils.c_file_manager as fm
```

```
>>> fm.expanduser('~')
'/home/username'
>>> fm.expanduser(fm.join('~','documents'))
'/home/username/documents'
```

secml.utils.c_file_manager.**dirsep**()
The character used by the operating system to separate pathname components. This is '/' for POSIX and '' for Windows. Note that knowing this is not sufficient to be able to parse or concatenate pathnames, use CFileManager.split() and CFileManager.join() instead, but it is occasionally useful.

secml.utils.c_file_manager.**get_tempfile**()
Returns an handle to a temporary file.

The file will be destroyed as soon as it is closed (including an implicit close when the object is garbage collected).

pickle_utils

secml.utils.pickle_utils.**save**(*file_path*, *obj*)
Save object to file using cPickle.

This functions stores a generic python object into a compressed gzip file (*.*gz*).

Saved objects can be loaded using *.load*.

Parameters

file_path [str] Path to destination file.

obj [object] Any python object to save.

Returns

obj_path [str] Full path to the stored object.

Notes

Objects are stored using **protocol 4** data stream format. For more information see https://docs.python.org/3/library/pickle.html#data-stream-format

secml.utils.pickle_utils.**load**(*file_path*, *encoding='bytes'*)
Load object from cPickle file.

Load a generic gzip compressed python object created by *.save*.

Parameters

file_path [str] Path to target file to read.

encoding [str, optional] Encoding to use for loading the file. Default 'bytes'.

download_utils

secml.utils.download_utils.**dl_file**(*url*, *output_dir*, *user=None*, *headers=None*, *chunk_size=1024*, *md5_digest=None*)

Download file from input url and store in output_dir.

Parameters

url [str] Url of the file to download.

output_dir [str] Path to the directory where the file should be stored. If folder does not exists, will be created.

user [str or None, optional] String with the user[:password] if required for accessing url.

headers [dict or None, optional] Dictionary with any additional header for the download request.

chunk_size [int, optional] Size of the data chunk to read from url in bytes. Default 1024.

md5_digest [str or None, optional] Expected MD5 digest of the downloaded file. If a different digest is computed, the downloaded file will be removed and ValueError is raised.

secml.utils.download_utils.**dl_file_gitlab**(*repo_url*, *file_path*, *output_dir*, *branch='master'*, *token=None*, *chunk_size=1024*, *md5_digest=None*)

Download file from a gitlab.com repository and store in output_dir.

Parameters

repo_url [str] Url of the repository from which download the file. Can include the *http(s)://gitlab.com/* prefix.

file_path [str] Path to the file to download, relative to the repository.

output_dir [str] Path to the directory where the file should be stored. If folder does not exists, will be created.

branch [str, optional] Branch from which the file should be downloaded. Default 'master'.

token [str or None, optional] Personal access token, required to access private repositories. See: https://docs.gitlab.com/ee/user/profile/personal_access_tokens.html

chunk_size [int, optional] Size of the data chunk to read from url in bytes. Default 1024.

md5_digest [str or None, optional] Expected MD5 digest of the downloaded file. If a different digest is computed, the downloaded file will be removed and ValueError is raised.

secml.utils.download_utils.**md5**(*fname*, *blocksize=65536*)

Generate RSA's MD5 digest for input file.

Parameters

fname [str] Path to the file to parse

blocksize [int] Size in bytes of the file chunks to read. Default 65536.

Returns

str MD5 hex digest of input file.

dict_utils

secml.utils.dict_utils.**load_dict**(*file_path*, *values_dtype=<class 'str'>*, *encoding='ascii'*)

Load dictionary from textfile.

Each file's line should be <key: value>

Parameters

file_path [str] Full path to the file to read.

values_dtype [dtype] Datatype of the values. Default str (string).

encoding [str, optional] Encoding to use for reading the file. Default 'ascii'.

Returns

dictionary [dict] Loaded dictionary with one key for each line in the input text file.

secml.utils.dict_utils.**merge_dicts**(**dicts*)

Shallow copy and merge any number of input dicts.

Precedence goes to key value pairs in latter dicts.

Parameters

dicts [dict1, dict2, ...] Any sequence of dict objects to merge.

Examples

```
>>> from secml.utils import merge_dicts
```

```
>>> d1 = {'attr1': 100, 'attr2': 200}
>>> d2 = {'attr3': 300, 'attr1': 999}  # Redefining `attr1`
```

```
>>> merge_dicts(d1, d2)  # Value of `attr1` will be set according to `d2`␣
↪dictionary
{'attr3': 300, 'attr2': 200, 'attr1': 999}
```

secml.utils.dict_utils.**invert_dict**(*d*)

Returns a new dict with keys as values and values as keys.

Parameters

d [dict] Input dictionary. If one value of the dictionary is a list or a tuple, each element of the sequence will be considered separately.

Returns

dict The new dictionary with d keys as values and d values as keys. In the case of duplicated d values, the value of the resulting key of the new dictionary will be a list with all the corresponding d keys.

Examples

```
>>> from secml.utils.dict_utils import invert_dict
```

```
>>> a = {'k1': 2, 'k2': 2, 'k3': 1}
>>> print(invert_dict(a))
{1: 'k3', 2: ['k1', 'k2']}
```

```
>>> a = {'k1': 2, 'k2': [2,3,1], 'k3': 1}
>>> print(invert_dict(a))
{1: ['k2', 'k3'], 2: ['k1', 'k2'], 3: 'k2'}
```

class secml.utils.dict_utils.**LastInDict**

Bases: collections.abc.MutableMapping

Last In Dictionary.

A standard dictionary that keeps in memory the key of the last set item. The setting behaviour is queue-like: a single element can be inserted in the dictionary each time.

The last key can be changes manually calling *LastInDict.lastitem_id = key*.

Examples

```
>>> from secml.utils import LastInDict
```

```
>>> li = LastInDict()
```

```
>>> li['key1'] = 123
>>> li['key2'] = 102030
```

```
>>> li.lastin_key
'key2'
>>> li.lastin
102030
```

Attributes

lastin

lastin_key

Methods

clear(self)	
get(self, key[, default])	
items(self)	
keys(self)	
pop(self, key[, default])	If key is not found, d is returned if given, otherwise KeyError is raised.
popitem(self)	as a 2-tuple; but raise KeyError if D is empty.
setdefault(self, key[, default])	

Continued on next page

Table 165 – continued from previous page

update(*args, **kwds)	If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v
values(self)	

property lastin

property lastin_key

class secml.utils.dict_utils.**SubLevelsDict**(*data*)

Bases: collections.abc.MutableMapping

Sub-Levels Dictionary.

A standard dictionary that allows easy access to attributes of contained objects at infinite deep.

Examples

```
>>> from secml.utils import SubLevelsDict
```

```
>>> class Foo:
...     attr2 = 5
```

```
>>> li = SubLevelsDict({'attr1': Foo()})
```

```
>>> print(type(li['attr1']))
<class 'dict_utils.Foo'>
>>> print(li['attr1.attr2'])
5
```

```
>>> li['attr1.attr2'] = 10  # Subattributes can be set in the same way
>>> print(li['attr1.attr2'])
10
```

Methods

clear(self)	
get(self, key[, default])	
items(self)	
keys(self)	
pop(self, key[, default])	If key is not found, d is returned if given, otherwise KeyError is raised.
popitem(self)	as a 2-tuple; but raise KeyError if D is empty.
setdefault(self, key[, default])	
update(*args, **kwds)	If E present and has a .keys() method, does: for k in E: D[k] = E[k] If E present and lacks .keys() method, does: for (k, v) in E: D[k] = v In either case, this is followed by: for k, v in F.items(): D[k] = v
values(self)	

list_utils

secml.utils.list_utils.**find_duplicates**(*l*)

Find and returns a python set with input list duplicates.

Parameters

l [list] List to examine.

Returns

duplicates [set] Python set with input list duplicates.

References

http://stackoverflow.com/questions/9835762/find-and-list-duplicates-in-python-list

Examples

```
>>> from secml.utils.list_utils import find_duplicates
>>> l = ['1', '1', 2, '3', 2]
>>> print(find_duplicates(l))
set(['1', 2])
```

mixed_utils

class secml.utils.mixed_utils.**AverageMeter**

Bases: object

Computes and stores the average and current value.

Attributes

val [float] Current value.

avg [float] Average.

sum [float] Cumulative sum of seen values.

count [int] Number of seen values.

Methods

update(self, val[, n])	Updated average and current value.

reset

reset(*self*)

update(*self*, *val*, *n=1*)

Updated average and current value.

Parameters

val [float] New current value.

n [int, optional] Multiplier for the current value. Indicates how many times the value should be counted in the average. Default 1.

class secml.utils.mixed_utils.**OrderedFlexibleClass**(**items*)

Bases: object

A flexible class exposing its attributes in a specific order when iterated.

Order of the attributes inside the class follows the inputs sequence. Any attribute set after class initialization will be placed at the end of attributes sequence (see examples).

Parameters

items [tuple1, tuple2, ...] Any custom sequence of tuples with the attributes to set. Each tuple must be a (key, value) pair.

Examples

```
>>> from secml.utils import OrderedFlexibleClass
```

```
>>> c = OrderedFlexibleClass(('attr1', None), ('attr2', 5))
>>> print(tuple(attr for attr in c))
(None, 5)
```

```
>>> c.attr3 = 123
>>> print(tuple(attr for attr in c))
(None, 5, 123)
```

Attributes

attr_order Returns a list specifing current attributes order.

property attr_order

Returns a list specifing current attributes order.

secml.utils.mixed_utils.**check_is_fitted**(*obj*, *attributes*, *msg=None*, *check_all=True*)

Check if the input object is trained (fitted).

Checks if the input object is fitted by verifying if all or any of the input attributes are not None.

Parameters

obj [object] Instance of the class to check. Must implement *.fit()* method.

attributes [str or list of str] Attribute or list of attributes to check. Es.: *['classes', 'n_features', ...], 'classes'*

msg [str or None, optional] If None, the default error message is: "this *{name}* is not trained. Call *.fit()* first.". For custom messages if '{name}' is present in the message string, it is substituted by the class name of the checked object.

check_all [bool, optional] Specify whether to check (True) if all of the given attributes are not None or (False) just any of them. Default True.

Raises

NotFittedError If *check_all* is True and any of the attributes is None; if *check_all* is False and all of attributes are None.

4.7.15 secml.settings

secml.settings.**SECML_HOME_DIR = '/home/docs/secml-data'**
Main directory for storing datasets, experiments, temporary files.

This is set by default to:

- Unix -> `${HOME}/secml-data`
- Windows -> `(${HOME}, ${USERPROFILE}, ${HOMEPATH}, ${HOMEDRIVE})/secml-data`

secml.settings.**SECML_CONFIG = ['/home/docs/secml-data/secml.conf', '/home/docs/checkouts/re**
Active *secml.conf* configuration files.

secml.settings.**SECML_DS_DIR = '/home/docs/secml-data/datasets'**
Main directory for storing datasets.

This is set by default to: `{SECML_HOME_DIR}/datasets`

secml.settings.**SECML_MODELS_DIR = '/home/docs/secml-data/models'**
Main directory where pre-trained models are stored.

This is set by default to: `{SECML_HOME_DIR}/models`

secml.settings.**SECML_EXP_DIR = '/home/docs/secml-data/experiments'**
Main directory of experiments data.

This is set by default to: `{SECML_HOME_DIR}/experiments`

secml.settings.**SECML_STORE_LOGS = False**
Whether to store logs to file. Default False.

secml.settings.**SECML_LOGS_DIR = '/home/docs/secml-data/logs'**
Directory where logs will be stored.

This is set by default to: `{SECML_HOME_DIR}/logs`

secml.settings.**SECML_LOGS_FILENAME = 'logs.log'**
Name of the logs file on disk. Default: *logs.log*.

secml.settings.**SECML_LOGS_PATH = '/home/docs/secml-data/logs/logs.log'**
Full path to the logs file: `{SECML_LOGS_DIR}/{SECML_LOGS_FILENAME}`.

secml.settings.**SECML_PYTORCH_USE_CUDA = True**
Controls if CUDA should be used by the PyTorch wrapper when available.

secml.settings.**parse_config**(*conf_files*, *section*, *parameter*, *default=None*, *dtype=None*)
Parse input *parameter* under *section* from configuration files.

Parameters file must have the following structure:

```
[section1]
param1=xxx
param2=xxx

[section2]
param1=xxx
param2=xxx
```

Parameters

conf_files [list] List with the paths of the configuration files to parse.

section [str] Section under which look for specified parameter.

parameter [str] Name of the parameter. This is not case-sensitive.

default [any, optional] Set default value of parameter. If None (default), parameter is considered required and so must be defined in the input configuration file. If not None, the value will be used if configuration file does not exists, section is not defined, or the parameter is not defined under section.

dtype [type or None, optional] Expected dtype of the parameter. If None (default), parameter will be parsed as a string. Other accepted values are: float, int, bool, str.

4.7.16 secml.testing

CUnitTest

class secml.testing.c_unittest.**CUnitTest**(*methodName='runTest'*)

Bases: unittest.case.TestCase

Superclass for unittests.

Provides a wrapper of *unittests.TestCase* in addition to:

- integrated logging functionalities (see *CLog*)
- integrated timing functionalities (see *CTimer*)
- addition assertion methods from *numpy.testing*
- *skip*, *skipif*, *importorskip* functions from *pytest*

Attributes

logger Logger for current object.

Methods

__call__(self, *args, **kwds)	Call self as a function.
addCleanup(self, function, *args, **kwargs)	Add a function, with arguments, to be called when the test is completed.
addTypeEqualityFunc(self, typeobj, function)	Add a type specific assertEqual style function to compare a type.
assertAlmostEqual(self, first, second[, ...])	Fail if the two objects are unequal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the difference between the two objects is more than the given delta.
assertCountEqual(self, first, second[, msg])	An unordered sequence comparison asserting that the same elements, regardless of order.
assertDictContainsSubset(self, subset, ...)	Checks whether dictionary is a superset of subset.
assertEqual(self, first, second[, msg])	Fail if the two objects are unequal as determined by the '==' operator.
assertFalse(self, expr[, msg])	Check that the expression is false.

Continued on next page

Table 168 – continued from previous page

assertGreater(self, a, b[, msg])	Just like self.assertTrue(a > b), but with a nicer default message.
assertGreaterEqual(self, a, b[, msg])	Just like self.assertTrue(a >= b), but with a nicer default message.
assertIn(self, member, container[, msg])	Just like self.assertTrue(a in b), but with a nicer default message.
assertIs(self, expr1, expr2[, msg])	Just like self.assertTrue(a is b), but with a nicer default message.
assertIsInstance(self, obj, cls[, msg])	Same as self.assertTrue(isinstance(obj, cls)), with a nicer default message.
assertIsNone(self, obj[, msg])	Same as self.assertTrue(obj is None), with a nicer default message.
assertIsNot(self, expr1, expr2[, msg])	Just like self.assertTrue(a is not b), but with a nicer default message.
assertIsNotNone(self, obj[, msg])	Included for symmetry with assertIsNone.
assertIsSubDtype(actual, desired)	Raises an AssertionError if the first dtype is NOT a typecode lower/equal in type hierarchy.
assertLess(self, a, b[, msg])	Just like self.assertTrue(a < b), but with a nicer default message.
assertLessEqual(self, a, b[, msg])	Just like self.assertTrue(a <= b), but with a nicer default message.
assertListEqual(self, list1, list2[, msg])	A list-specific equality assertion.
assertLogs(self[, logger, level])	Fail unless a log message of level *level* or higher is emitted on *logger_name* or its children.
assertMultiLineEqual(self, first, second[, msg])	Assert that two multi-line strings are equal.
assertNotAlmostEqual(self, first, second[, ...])	Fail if the two objects are equal as determined by their difference rounded to the given number of decimal places (default 7) and comparing to zero, or by comparing that the difference between the two objects is less than the given delta.
assertNotEqual(self, first, second[, msg])	Fail if the two objects are equal as determined by the '!=' operator.
assertNotIn(self, member, container[, msg])	Just like self.assertTrue(a not in b), but with a nicer default message.
assertNotIsInstance(self, obj, cls[, msg])	Included for symmetry with assertIsInstance.
assertNotRegex(self, text, unexpected_regex)	Fail the test if the text matches the regular expression.
assertRaises(self, expected_exception, ...)	Fail unless an exception of class expected_exception is raised by the callable when invoked with specified positional and keyword arguments.
assertRaisesRegex(self, expected_exception, ...)	Asserts that the message in a raised exception matches a regex.
assertRegex(self, text, expected_regex[, msg])	Fail the test unless the text matches the regular expression.
assertSequenceEqual(self, seq1, seq2[, msg, ...])	An equality assertion for ordered sequences (like lists and tuples).
assertSetEqual(self, set1, set2[, msg])	A set-specific equality assertion.
assertTrue(self, expr[, msg])	Check that the expression is true.
assertTupleEqual(self, tuple1, tuple2[, msg])	A tuple-specific equality assertion.

Continued on next page

Table 168 – continued from previous page

assertWarns(self, expected_warning, *args, ...)	Fail unless a warning of class warnClass is triggered by the callable when invoked with specified positional and keyword arguments.
assertWarnsRegex(self, expected_warning, ...)	Asserts that the message in a triggered warning matches a regexp.
assert_allclose(self, actual, desired[, ...])	Raises an AssertionError if two objects are not equal up to desired tolerance.
assert_approx_equal(self, actual, desired[, ...])	Raises an AssertionError if two items are not equal up to significant digits.
assert_array_almost_equal(self, x, y[, ...])	Raises an AssertionError if two objects are not equal up to desired precision.
assert_array_almost_equal_nulp(self, x, y[, ...])	Compare two arrays relatively to their spacing.
assert_array_equal(self, x, y[, err_msg, ...])	Raises an AssertionError if two array_like objects are not equal.
assert_array_less(self, x, y[, err_msg, verbose])	Raises an AssertionError if two array_like objects are not ordered by less than.
assert_array_max_ulp(self, a, b[, maxulp, dtype])	Check that all items of arrays differ in at most N Units in the Last Place.
debug(self)	Run the test without collecting errors in a TestResult
doCleanups(self)	Execute all cleanup functions.
fail(self[, msg])	Fail immediately, with the given message.
failureException	alias of builtins.AssertionError
main	alias of unittest.main.TestProgram
setUp(self)	Hook method for setting up the test fixture before exercising it.
setUpClass()	Hook method for setting up class fixture before running tests in the class.
shortDescription(self)	Returns a one-line description of the test, or None if no description has been provided.
skipTest(self, reason)	Skip this test.
subTest(self[, msg])	Return a context manager that will return the enclosed block of code in a subtest identified by the optional message and keyword parameters.
tearDown(self)	Hook method for deconstructing the test fixture after testing it.
tearDownClass()	Hook method for deconstructing the class fixture after running all tests in the class.
timer(self)	Returns a CTimer to be used as context manager.

assertAlmostEquals	
assertDictEqual	
assertEquals	
assertNotAlmostEquals	
assertNotEquals	
assertNotRegexpMatches	
assertRaisesRegexp	
assertRegexpMatches	
assert_	
countTestCases	
defaultTestResult	
failIf	
failIfAlmostEqual	
failIfEqual	
failUnless	
failUnlessAlmostEqual	
failUnlessEqual	
failUnlessRaises	
id	
importskip	
run	
skip	
skipif	

static assertIsSubDtype(*actual*, *desired*)

Raises an AssertionError if the first dtype is NOT a typecode lower/equal in type hierarchy.

Parameters

actual [dtype or str] dtype or string representing a typecode.

desired [dtype or str] dtype or string representing a typecode.

assert_allclose(*self*, *actual*, *desired*, *rtol=1e-06*, *atol=0*, *equal_nan=True*, *err_msg=''*, *verbose=True*)

Raises an AssertionError if two objects are not equal up to desired tolerance.

The test is equivalent to `allclose(actual, desired, rtol, atol)` (note that `allclose` has different default values). It compares the difference between *actual* and *desired* to `atol + rtol * abs(desired)`.

New in version 1.5.0.

Parameters

actual [array_like] Array obtained.

desired [array_like] Array desired.

rtol [float, optional] Relative tolerance.

atol [float, optional] Absolute tolerance.

equal_nan [bool, optional.] If True, NaNs will compare equal.

err_msg [str, optional] The error message to be printed in case of failure.

verbose [bool, optional] If True, the conflicting values are appended to the error message.

Raises

AssertionError If actual and desired are not equal up to specified precision.

See also:

assert_array_almost_equal_nulp, ***assert_array_max_ulp***

Examples

```
>>> x = [1e-5, 1e-3, 1e-1]
>>> y = np.arccos(np.cos(x))
>>> np.testing.assert_allclose(x, y, rtol=1e-5, atol=0)
```

assert_approx_equal(*self*, *actual*, *desired*, *significant=6*, *err_msg=''*, *verbose=True*)
Raises an AssertionError if two items are not equal up to significant digits.

Note: It is recommended to use one of *assert_allclose*, *assert_array_almost_equal_nulp* or *assert_array_max_ulp* instead of this function for more consistent floating point comparisons.

Given two numbers, check that they are approximately equal. Approximately equal is defined as the number of significant digits that agree.

Parameters

actual [scalar] The object to check.

desired [scalar] The expected object.

significant [int, optional] Desired precision, default is 7.

err_msg [str, optional] The error message to be printed in case of failure.

verbose [bool, optional] If True, the conflicting values are appended to the error message.

Raises

AssertionError If actual and desired are not equal up to specified precision.

See also:

assert_allclose Compare two array_like objects for equality with desired relative and/or absolute precision.

assert_array_almost_equal_nulp, ***assert_array_max_ulp***, **assert_equal**

Examples

```
>>> np.testing.assert_approx_equal(0.12345677777777e-20, 0.1234567e-20)
>>> np.testing.assert_approx_equal(0.12345670e-20, 0.12345671e-20,
...                                significant=8)
>>> np.testing.assert_approx_equal(0.12345670e-20, 0.12345672e-20,
...                                significant=8)
Traceback (most recent call last):
    ...
AssertionError:
Items are not equal to 8 significant digits:
 ACTUAL: 1.234567e-21
 DESIRED: 1.2345672e-21
```

the evaluated condition that raises the exception is

```
>>> abs(0.12345670e-20/1e-21 - 0.12345672e-20/1e-21) >= 10**-(8-1)
True
```

assert_array_almost_equal(*self*, *x*, *y*, *decimal=6*, *err_msg=''*, *verbose=True*)
Raises an AssertionError if two objects are not equal up to desired precision.

Note: It is recommended to use one of *assert_allclose*, *assert_array_almost_equal_nulp* or *assert_array_max_ulp* instead of this function for more consistent floating point comparisons.

The test verifies identical shapes and that the elements of `actual` and `desired` satisfy.

```
abs(desired-actual) < 1.5 * 10**(-decimal)
```

That is a looser test than originally documented, but agrees with what the actual implementation did up to rounding vagaries. An exception is raised at shape mismatch or conflicting values. In contrast to the standard usage in numpy, NaNs are compared like numbers, no assertion is raised if both objects have NaNs in the same positions.

Parameters

x [array_like] The actual object to check.

y [array_like] The desired, expected object.

decimal [int, optional] Desired precision, default is 6.

err_msg [str, optional] The error message to be printed in case of failure.

verbose [bool, optional] If True, the conflicting values are appended to the error message.

Raises

AssertionError If actual and desired are not equal up to specified precision.

See also:

assert_allclose Compare two array_like objects for equality with desired relative and/or absolute precision.

assert_array_almost_equal_nulp, ***assert_array_max_ulp***, **assert_equal**

Examples

the first assert does not raise an exception

```
>>> np.testing.assert_array_almost_equal([1.0,2.333,np.nan],
...                                      [1.0,2.333,np.nan])
```

```
>>> np.testing.assert_array_almost_equal([1.0,2.33333,np.nan],
...                                      [1.0,2.33339,np.nan], decimal=5)
Traceback (most recent call last):
    ...
AssertionError:
Arrays are not almost equal to 5 decimals

Mismatched elements: 1 / 3 (33.3%)
Max absolute difference: 6.e-05
```

(continues on next page)

(continued from previous page)

```
Max relative difference: 2.57136612e-05
 x: array([1.     , 2.33333,     nan])
 y: array([1.     , 2.33339,     nan])
```

```
>>> np.testing.assert_array_almost_equal([1.0,2.33333,np.nan],
...                                      [1.0,2.33333, 5], decimal=5)
Traceback (most recent call last):
    ...
AssertionError:
Arrays are not almost equal to 5 decimals

x and y nan location mismatch:
 x: array([1.     , 2.33333,     nan])
 y: array([1.     , 2.33333, 5.     ])
```

assert_array_almost_equal_nulp(*self*, *x*, *y*, *nulp=1*)

Compare two arrays relatively to their spacing.

This is a relatively robust method to compare two arrays whose amplitude is variable.

Parameters

x, y [array_like] Input arrays.

nulp [int, optional] The maximum number of unit in the last place for tolerance (see Notes). Default is 1.

Returns

None

Raises

AssertionError If the spacing between *x* and *y* for one or more elements is larger than *nulp*.

See also:

assert_array_max_ulp Check that all items of arrays differ in at most N Units in the Last Place.

spacing Return the distance between x and the nearest adjacent number.

Notes

An assertion is raised if the following condition is not met:

```
abs(x - y) <= nulps * spacing(maximum(abs(x), abs(y)))
```

Examples

```
>>> x = np.array([1., 1e-10, 1e-20])
>>> eps = np.finfo(x.dtype).eps
>>> np.testing.assert_array_almost_equal_nulp(x, x*eps/2 + x)
```

```
>>> np.testing.assert_array_almost_equal_nulp(x, x*eps + x)
Traceback (most recent call last):
  ...
AssertionError: X and Y are not equal to 1 ULP (max is 2)
```

assert_array_equal(*self*, *x*, *y*, *err_msg=''*, *verbose=True*)

Raises an AssertionError if two array_like objects are not equal.

Given two array_like objects, check that the shape is equal and all elements of these objects are equal (but see the Notes for the special handling of a scalar). An exception is raised at shape mismatch or conflicting values. In contrast to the standard usage in numpy, NaNs are compared like numbers, no assertion is raised if both objects have NaNs in the same positions.

The usual caution for verifying equality with floating point numbers is advised.

Parameters

x [array_like] The actual object to check.

y [array_like] The desired, expected object.

err_msg [str, optional] The error message to be printed in case of failure.

verbose [bool, optional] If True, the conflicting values are appended to the error message.

Raises

AssertionError If actual and desired objects are not equal.

See also:

assert_allclose Compare two array_like objects for equality with desired relative and/or absolute precision.

assert_array_almost_equal_nulp, ***assert_array_max_ulp***, **assert_equal**

Notes

When one of *x* and *y* is a scalar and the other is array_like, the function checks that each element of the array_like object is equal to the scalar.

Examples

The first assert does not raise an exception:

```
>>> np.testing.assert_array_equal([1.0,2.33333,np.nan],
...                               [np.exp(0),2.33333, np.nan])
```

Assert fails with numerical imprecision with floats:

```
>>> np.testing.assert_array_equal([1.0,np.pi,np.nan],
...                               [1, np.sqrt(np.pi)**2, np.nan])
Traceback (most recent call last):
    ...
AssertionError:
Arrays are not equal

Mismatched elements: 1 / 3 (33.3%)
Max absolute difference: 4.4408921e-16
Max relative difference: 1.41357986e-16
 x: array([1.      , 3.141593,      nan])
 y: array([1.      , 3.141593,      nan])
```

Use *assert_allclose* or one of the nulp (number of floating point values) functions for these cases instead:

```
>>> np.testing.assert_allclose([1.0,np.pi,np.nan],
...                            [1, np.sqrt(np.pi)**2, np.nan],
...                            rtol=1e-10, atol=0)
```

As mentioned in the Notes section, *assert_array_equal* has special handling for scalars. Here the test checks that each value in *x* is 3:

```
>>> x = np.full((2, 5), fill_value=3)
>>> np.testing.assert_array_equal(x, 3)
```

assert_array_less(*self*, *x*, *y*, *err_msg=''*, *verbose=True*)

Raises an AssertionError if two array_like objects are not ordered by less than.

Given two array_like objects, check that the shape is equal and all elements of the first object are strictly smaller than those of the second object. An exception is raised at shape mismatch or incorrectly ordered values. Shape mismatch does not raise if an object has zero dimension. In contrast to the standard usage in numpy, NaNs are compared, no assertion is raised if both objects have NaNs in the same positions.

Parameters

x [array_like] The smaller object to check.

y [array_like] The larger object to compare.

err_msg [string] The error message to be printed in case of failure.

verbose [bool] If True, the conflicting values are appended to the error message.

Raises

AssertionError If actual and desired objects are not equal.

See also:

assert_array_equal tests objects for equality

assert_array_almost_equal test objects for equality up to precision

Examples

```
>>> np.testing.assert_array_less([1.0, 1.0, np.nan], [1.1, 2.0, np.nan])
>>> np.testing.assert_array_less([1.0, 1.0, np.nan], [1, 2.0, np.nan])
Traceback (most recent call last):
    ...
AssertionError:
Arrays are not less-ordered

Mismatched elements: 1 / 3 (33.3%)
Max absolute difference: 1.
Max relative difference: 0.5
 x: array([ 1.,  1., nan])
 y: array([ 1.,  2., nan])
```

```
>>> np.testing.assert_array_less([1.0, 4.0], 3)
Traceback (most recent call last):
    ...
AssertionError:
Arrays are not less-ordered
```

(continues on next page)

(continued from previous page)

```
Mismatched elements: 1 / 2 (50%)
Max absolute difference: 2.
Max relative difference: 0.66666667
 x: array([1., 4.])
 y: array(3)
```

```
>>> np.testing.assert_array_less([1.0, 2.0, 3.0], [4])
Traceback (most recent call last):
    ...
AssertionError:
Arrays are not less-ordered

(shapes (3,), (1,) mismatch)
 x: array([1., 2., 3.])
 y: array([4])
```

assert_array_max_ulp (*self*, *a*, *b*, *maxulp=1*, *dtype=None*)

Check that all items of arrays differ in at most N Units in the Last Place.

Parameters

a, b [array_like] Input arrays to be compared.

maxulp [int, optional] The maximum number of units in the last place that elements of *a* and *b* can differ. Default is 1.

dtype [dtype, optional] Data-type to convert *a* and *b* to if given. Default is None.

Returns

ret [ndarray] Array containing number of representable floating point numbers between items in *a* and *b*.

Raises

AssertionError If one or more elements differ by more than *maxulp*.

See also:

assert_array_almost_equal_nulp Compare two arrays relatively to their spacing.

Notes

For computing the ULP difference, this API does not differentiate between various representations of NAN (ULP difference between 0x7fc00000 and 0xffc00000 is zero).

Examples

```
>>> a = np.linspace(0., 1., 100)
>>> res = np.testing.assert_array_max_ulp(a, np.arcsin(np.sin(a)))
```

importskip

property logger
Logger for current object.

main
alias of `unittest.main.TestProgram`

classmethod setUpClass()
Hook method for setting up class fixture before running tests in the class.

skip

skipif

timer(*self*)
Returns a CTimer to be used as context manager.

4.7.17 UPDATE GUIDES

From 0.8.* to 0.9

Make sure you view this update guide from the tag (version) of SecML you would like to install. In most cases this should be the highest numbered production tag (without rc in it).

To update the current `v0.8.*` version to `v0.9`, run the following steps:

1. Configuration file

The following new configuration settings are now available. Please update your `$SECML_HOME_DIR/secml.conf` file if needed, otherwise default values will be used.

1. Added new section `[secml:pytorch]` to control the behaviour of the classes related to `pytorch` library support. The following options are available:
   - `data_dir`. Directory for storing `pytorch` data. Default: `SECML_HOME_DIR/pytorch-data`.
   - `use_cuda`. True (default) if CUDA should be used by the `pytorch` wrapper.

2. Deprecations

The following classes, methods or functions are now deprecated.

- `CClassifierKDE`: use `CClassifierSkLearn` with `sklearn.neighbors.KernelDensity` instead
- `CClassifierMCSLinear`: use `CClassifierSkLearn` with `sklearn.ensemble.BaggingClassifier` instead
- `CPreProcess.revert()`: use `.inverse_transform()` method instead

4.7.18 CHANGELOG

v0.13 (24/07/2020)

- #814 Added new evasion attack `CAttackEvasionPGDExp`.
- #780 Added new classifier `CClassifierDNR` implementing Deep Neural Rejection (DNR). See *Sotgiu et al. "Deep neural rejection against adversarial examples", EURASIP J. on Info. Security (2020).*
- #47 Added new classifier `CClassifierMulticlassOVO` implementing One-vs-One multiclass classification scheme.
- #765 Extended `CModule` to support trainable modules via `fit` and `fit_forward` functions.
- #800 Security evaluation can now be run using Cleverhans attacks. The name of the parameter to check should be specified as `attack_params.<param_name>` as an input argument for the constructor of `CSecEval`.
- #839 Experimental support of Windows operating system (version 7 or later).

Requirements (1 change)

- #768 Removed temporary pin of Pillow to v6 which used to break torch and torchvision packages.

Added (4 changes)

- #100007 Added new experimental package `ml.scalers` with a different implementation of `ml.features.normalization` classes directly based Scikit-Learn's scalers. Included classes are: `CScalerMinMax`, `CScalerStd`, `CScalerNorm`.
- #770 Added new methods to convert a `CArray` to specific `scipy.sparse` array formats: `tocoo`, `tocsc`, `todia`, `todok`, `tolil`.
- #812 `CAttackPoisoning` now exposes: `x0`, `xc`, `yc`, `objective_function` and `objective_function_gradient`.
- #776 `n_jobs` is now a init parameter of `CModule` and subclasses and not passed via `fit` anymore.

Improved (12 changes)

- #817 Added `CClassifierSVM` native support to OVA multiclass scheme, without replicating the kernel in each one-vs-all classifier.
- #574 Added `_clear_cache` mechanism to `CModule` and classes that require caching data in the forward pass before backward (e.g., exponential kernels do that to avoid re-computing the kernel matrix in the backward pass).
- #820 Add parallel execution of `forward` method for `CClassifierMulticlassOVA` and `CClassifierMulticlassOVO`.
- #815 Simplified `CAttack` interface (now only requires implementing `run` as required by `CSecEval`).
- #574 Modified kernel and classifier interfaces to allow their use as preprocessing modules.
- #775 Improved efficiency in gradient computation of SVMs, by back-propagating the alpha values to the kernel.
- #773 Improved efficiency in the computation of gradients of evasion attacks (`CAttackEvasionPGDLS`). Now gradient is called once rather than twice to compute the gradient of the objective function.

- #801 `CSecEval` will now check that the `param_name` input argument can be found in the attack class used in the evaluation.
- #695 `COptimizerPGD` now exits optimization if constraint radius is 0. `COptimizerPGD`, `COptimizerPGDLS` and `COptimizerPGDExp` will now raise a warning if the 0-radius constraint is defined outside the given bounds.
- #828 `CClassifierSVM` now uses `n_jobs` parameter for parallel execution of training in case of multiclass datasets.
- #767 Using `scipy.sparse .hstack` and `.vstack` instead of a custom implementation in `CSparse.concatenate`.
- #772 Using `scipy.sparse .argmin` and `.argmax` instead of a custom implementation in `CSparse.argmin` and `CSparse.argmax`.

Changed (6 changes)

- #817 Kernel is now used as preprocess in `CClassifierSVM`.
- #817 Removed `store_dual_vars` and `kernel.setter` from `CClassifierSVM`. Now a linear SVM is trained in the primal (w,b) if `kernel=None`, otherwise it is trained in the dual (alpha and b), on the precomputed training kernel matrix.
- #765 Unified `fit` interface from `fit(ds)` to `fit(x,y)` to be consistent across normalizers and classifiers.
- #574 Removed redundant definitions of `gradient(x, w)` from `CKernelRBF`, `CKernelLaplacian`, `CKernelEuclidean`, `CClassifierDNN`, `CNormalizerUnitNorm`. The protected property `grad_requires_forward` now specifies if gradient has to compute an explicit forward pass or only propagate the input `x` through the pre-processing chain before calling `backward`.
- #823 Removed `surrogate_data` parameter from `CAttackPoisoning` and renamed it to `double_init_ds` in `CAttackEvasion` subclasses.
- #829 `CClassifierRejectThreshold` now returns wrapped classifier classes plus the reject class (-1).

Fixed (10 changes)

- #816 Fixed stop condition of `COptimizerPGD` which was missing index `i`.
- #825 Infer the number of attacked classifier classes directly from it (instead of inferring it from surrogate data) in `CAttackEvasionPGDLS` to fix a crash when the class index of data points is greater or equal than the number of alternative data points.
- #810 Fixed `CClassifierPyTorch.backward` not working properly due to a miscalculation of the number of input features of the model when a `CNormalizeDNN` is used as preprocessor.
- #803 Fixed checks on the inner classifier in `CClassifierRejectThreshold` which can be bypassed by using the clf attribute setter, now removed.
- #818 Fixed `CCreator.set` not allowing to set writable attributes of level-0 readable-only attributes.
- #819 Fixed `CCreator.get_params` not returning level-0 not-writable attributes having one or more writable attributes.
- #785 Fixed constant override of matplotlib backend in `CFigure` on Windows systems.
- #783 Fixed `model_zoo.load_model` improperly building download urls depending on the system default url separator.

- #771 Fixed the following methods of `CSparse` to ensure they properly work independently from the sparse array format: `save`, `load`, `__pow__`, `round`, `nan_to_num`, `logical_and`, `unique`, `bincount`, `prod`, `all`, `any`, `min`, `max`.
- #769 `CArray.tocsr()` now always returns a `scipy.sparse.csr_matrix` array as expected.

Removed & Deprecated (2 changes)

- #540 Removed `discrete` and `surrogate_classifier` parameter from `CAttack`.
- #777 Deprecated attribute `kernel` is now removed from `CClassifierSGD`, `CClassifierRidge` and `CClassifierLogistic` classifiers.

Documentation (10 changes)

- #839 Windows is now displayed as a supported Operating System in README and setup.
- #806 Documented pytorch extra component installation requirements under Windows.
- #834 Temporarily pinned `numpydoc` to `< 1.1` to avoid compatibility issues of the newest version.
- #807 Documentation is now built using Sphinx https://readthedocs.org/ theme v0.5 or higher.
- #830 Fixed links to repository pages by adding a dash after project name.
- #758 Added a direct link to the gitlab.com repository in README.
- #788 Notebooks now include a warning about the required extra components (if any).
- #787 Fixed argmin -> argmax typo in docstring of `CClassifierRejectThreshold.predict` method.
- #789 Fixed notebook 4 not correctly generating a separate dataset for training the target classifiers.
- #791 Fixed `random_state` not set for `CClassifierDecisionTree` in notebook 4.

v0.12 (11/03/2020)

- #726 Refactored kernel package (now `secml.ml.kernels`). Kernel classes are now inherited from `CModule`, which enables computing gradients more efficiently. This will enable us to use kernels as preprocessors in future releases.
- #755 Package `secml.ml.model_zoo` has been moved to `secml.model_zoo`.
- #721 Dictionary with model zoo definitions is now dynamically downloaded and updated from our repository located at https://gitlab.com/secml/secml-zoo. The package `model_zoo.models` containing python scripts defining models structure is now removed and the scripts will be downloaded from the same repository upon request.

Added (7 changes)

- #660 `CClassifierPyTorch` now accepts as input a PyTorch learning rate scheduler via the `optimizer_scheduler` parameter.
- #678 Added new parameter `return_optimizer` to `CClassifierPyTorch.get_state` which allows getting the state of the classifier without including the state of the `optimizer` (and of the new `optimizer_scheduler`).
- Added `random_state` parameter to `CPSKMedians`.
- Added the parameter `minlength` to the `bincount` method of `CArray`.
- Added new `CNormalizerTFIDF` which implements a term frequency–inverse document frequency features normalizer.
- #666 Added new `utils.download_utils.dl_file_gitlab` function which allows downloading a file from a gitlab.com repository, including branch and access token setting.
- #722 Added new optional parameter `headers` to `utils.download_utils.dl_file` function which allows specifying additional headers for the download request.

Improved (8 changes)

- #664 The following `CClassifierPyTorch` parameters can now be modified after instancing the class: `optimizer`, `epochs`, `batch_size`. This will make some procedures easier, like fine-tuning a pre-trained network.
- #712 `download_utils.dl_file()` will now use the filename stored in response's headers if available. The previous behavior (get the last part of the download url) will be used as a fallback.
- #748 `CNormalizerUnitNorm` re-factored by adding gradient computation.
- #706 Rewrite `CKernelRBF` gradient when passing `w` to speed up computations by avoiding broadcasting.
- #730 `CClassifierPyTorch` has been modified to clean cached outputs and save memory when caching such data is not required.
- Internally optimized variables can be stored inside the attack class and fetched when needed.
- Accurate evaluation of objective function for some cleverhans attacks (CW, Elastic Net).
- #666 Model zoo downloader `ml.model_zoo.load_model` function will now try to download the version of a requested model corresponding to the version of secml. If not found, the latest 'master' version of the model will be downloaded instead.

Changed (3 changes)

- #664 When passing pre-trained models to `CClassifierDNN` and subclasses the new `pretrained` parameter should now be set to `True`. Optionally, an array of classes on which the model has been pre-trained can be passed via the new `pretrained_classes` parameter. If `pretrained_classes` is left `None`, the number of classes will be inferred from the size of the last DNN layer as before.
- `CConstraintL2.project(x)` projects now `x` onto a hypersphere of radius `radius-tol`, with `tol=1e-6`. This conservative projection ensures that `x` is projected always inside the hypersphere, overcoming projection violations due to numerical rounding errors.
- `CModule.gradient` is not calling `forward` anymore, but only prepares data for `backward`. The forward step is not required, indeed, for modules that implement analytical gradients rather than autodiff.

Fixed (10 changes)

- #677 Fixed `CClassifierPyTorch.get_state` crashing when optimizer is not defined.
- #134 Fixed passage of `n_jobs` parameter to `CDataLoaderPyTorch` in `CClassifierPyTorch` where 2 processes are being used by the loader even if `n_jobs` is set to 1. The default value for parameter `num_workers` in `CDataLoaderPyTorch` is now correctly 0.
- #749 Fixed `CArray.argmin` and `.argmax` returning float types when applied to sparse arrays of float dtype.
- Gradient is now correctly computed in `CClassifierPytorch` even if `softmax_outputs` are active.
- #707 Fixed initial value of the objective function being computed before starting point projection in `COptimizerPGDLS`.
- #667 Fixed `download_utils.dl_file()` not removing url parameters from the name of the stored file.
- #715 `download_utils.dl_file()` now correctly manage the absence of the 'content-length' header from response.
- Inverted sign of computed kernel similarity (to have a distance measure).
- #710 Random seed in `CClassifierPyTorch` is now correctly applied also when running on the CuDNN backend.
- #639: Objective function parameter (`objective_function`) in `CAttackEvasionCleverhans` is now correctly populated for `ElasticNetMethod` and `SPSA` attacks.

Removed & Deprecated (5 changes)

- #748 `CNormalizerUnitNorm.inverse_transform` has been removed (it only worked if one inverted `x` after transforming it, but not if other transforms were applied in between).
- Removed the parameters `n_feats` and `n_classes` from the interface of `CAttackEvasionCleverhans`.
- #744 Deprecate kernel parameter from `CClassifierSGD` and `CClassifierRidge` and removed deprecated parameter `kernel='linear'` from notebook `01-Training.ipynb`.
- #643 Removed deprecated parameter `random_seed` from `CClassifierLogistic`. Use `random_state` instead.
- #643 Removed deprecated method `is_linear` from `CClassifier`, `CNormalizer`, and related subclasses.

Documentation (5 changes)

- #756 Fixed format of output arrays reported in `CArray.__mul__` and `.__truediv__` methods.
- #681 Fixed few typos in `CExplainerIntegratedGradients`.
- #674 Added `CClassifierDNN` to the documentation.
- #711 Added a "How to cite SecML" section in README.
- #703 Updated copyright notice in README.

v0.11.2 (07/01/2020)

- This version brings fixes for a few reported issues with `CAttack` and subclasses, along with the new Developers and Contributors guide.

Requirements (1 change)

- #700 Temporarily pinned `Pillow` to v6 to avoid breaking `torch` and `torchvision` packages.

Fixed (7 changes)

- #698 Fixed `CAttackEvasionCleverhans` definition of `class_type`.
- #662 The number of function and gradient evaluations made during double initialization in `CAttackEvasionPGDLS` are now correctly considered by `.f_eval` and `.grad_eval` properties.
- #699 Fixed batch processing in `CClassifierPyTorch` not working properly if the number of samples to be classified is not a multiple of `batch_size`.
- #691 Function and gradient evaluation counts in `CAttackEvasionCleverhans` returned by `.f_eval` and `.grad_eval` properties now only consider the last optimized sample, consistently with other `CAttack` subclasses.
- #701 Default value of `double_init` parameter in `CAttackEvasionPGDLS` set to True as originally intended.
- #684 The solution returned by `COptimizerPGD` is now always the best one found during the minimization process.
- #697 Fixed unittests failing under numpy v1.18 due to a change in the errors raised by `genfromtxt`.

Documentation (2 changes)

- #671 Added Developers and Contributors guide.
- #694 Added a new notebook tutorial on advanced evasion attacks using Deep Neural Networks and ImageNet dataset.

v0.11.1 (18/12/2019)

- Fixed compatibility issues with recently released scikit-learn v0.22 and scipy v1.4.

Fixed (3 changes)

- #687 Fixed reshaping of sparse arrays to vector-like when using Scipy v1.4.
- #686 Replaced deprecated import of `interp` function from scipy namespace instead of numpy namespace.
- #668 Fixed unittests failing under scikit-learn v0.22 due to a change in their class output.

v0.11 (02/12/2019)

- #653 Added new `secml.ml.model_zoo` package, which provides a zoo of pre-trained SecML models. The list of available models will be greatly expanded in the future. See https://secml.gitlab.io/secml.ml.model_zoo.html for more details.
- #629 Greatly improved the performance of the `grad_f_x` method for `CClassifier` and `CPreProcess` classes, especially when nested via `preprocess` attribute.
- #613 Support for Python 2.7 is dropped. Python version 3.5, 3.6, or 3.7 is now required.

Requirements (2 changes)

- #633 The following dependencies are now required: `numpy >= 1.17`, `scipy >= 1.3.1`, `scikit-learn >= 0.21` `matplotlib = 3`.
- #622 Removed dependency on `six` library.

Added (5 changes)

- #539 Added new core interface to get and set the state of an object instance: `set_state`, `get_state`, `save_state`, `load_state`. The state of an object is a simple human-readable Python dictionary object which stores the data necessary to restore an instance to a specific status. Please not that to guarantee the exact match between the original object instance and the restored one, the standard save/load interface should be used.
- #647 Added new function `core.attr_utils.get_protected` which returns a protected attribute from a class (if exists).
- #629 `CClassifier` and `CPreProcess` classes now provide a `gradient` method, which computes the gradient by doing a forward and a backward pass on the classifier or preprocessor function chain, accepting an optional pre-multiplier `w`.
- #539 Added new accessible attributes to multiple classes: `CNormalizerMinMax .m .q`; `CReducerLDA .lda`; `CClassifierKNN .tr`; `CClassifierRidge .tr`; `CClassifierSGD .tr`; `CClassifierPyTorch .trained`.
- #640 Added `random_state` parameter to `CClassifierDecisionTree`.

Improved (6 changes)

- #631 Data objects are now stored using protocol 4 by `pickle_utils.save`. This protocol adds support for very large objects, pickling more kinds of objects, and some data format optimizations.
- #639 Objective function parameter (`objective_function`) in `CAttackEvasionCleverhans` is now correctly populated for the following attacks: `CarliniWagnerL2`, `FastGradientMethod`, `ProjectedGradientDescent`, `LBFGS`, `MomentumIterativeMethod`, `MadryEtAl`, `BasicIterativeMethod`.
- #638 The sequence of modifications to the attack point (`x_seq` parameter) is now correctly populated in `CAttackEvasionCleverhans`.
- #595 A pre-trained classifier can now be passed to `CClassifierRejectThreshold` to avoid running fit twice.
- #627 Slight improvement of `CKernel.gradient()` method performance by removing unnecessary calls.
- #630 Sparse data can now be used in `CKernelHistIntersect`.

Changed (2 changes)

- #616 Renamed `CModelCleverhans` to `_CModelCleverhans` as this class is not supposed to be explicitly used.
- #111 Default value of the parameter `tol` changed from `-inf` to `None` in `CClassifierSGD`. This change should not alter the classifier behavior when using the default parameters.

Fixed (8 changes)

- #611 Fixed `CDataloaderMNIST` crashing depending on the desired number of samples and digits to load.
- #652 Number of gradient computations returned by `CAttackEvasionCleverhans.grad_eval` is now accurate.
- #650 Fixed `CAttackEvasionCleverhans.f_eval` wrongly returns the number of gradient evaluations.
- #637 Fixed checks on `y_taget` in `CAttackEvasionCleverhans` which compared the 0 label to untargeted case (`y_true = None`).
- #648 Function `core.attr_utils.is_public` now correctly return False for properties.
- #649 Fixed wrong use of `core.attr_utils.is_public` in `CCreator` and `CDatasetHeader`.
- #655 Fixed `CClassifierRejectThreshold.n_classes` not taking into account the rejected class (label -1).
- #636 Fixed a `TypeError` raised by `CFigure.clabel()` when using matplotlib 3.

Removed & Deprecated (4 changes)

- #628 Method `is_linear` of `CClassifier` and `CNormalizer` subclasses is now deprecated.
- #641 Parameter `random_seed` of `CClassifierLogistic` is now deprecated. Use `random_state` instead.
- #603 Removed deprecated class `CNormalizerMeanSTD`.
- #603 Removed deprecated parameter `batch_size` from `CKernel` and subclasses.

Documentation (4 changes)

- #625 Reorganized notebooks tutorials into different categories: *Machine Learning*, *Adversarial Machine Learning*, and *Explainable Machine Learning*.
- #615 Added a tutorial notebook on the use of Cleverhans library wrapper.
- #607 Settings module `secml.settings` is now correctly displayed in the docs.
- #626 Added missing reference to `CPlotMetric` class in docs.

v0.10 (29/10/2019)

- #535 Added new package `secml.explanation`, which provides different methods for explaining machine learning models. See documentation and examples for more information.
- #584 **[beta]** Added `CAttackEvasionCleverhans` to support adversarial attacks from CleverHans, a Python library to benchmark vulnerability of machine learning systems to adversarial examples.

Requirements (1 change)

- #580 PyTorch version `1.3` is now supported.

Added (4 changes)

- #565 Added new abstract interface `CClassifierDNN` from which new classes implementing Deep Neural Networks can inherit.
- #555 Added `CNormalizerDNN`, which allows using a `CClassifierDNN` as a preprocessor.
- #593 Added `CDataLoaderTorchDataset`, which allows converting a `torchvision` dataset into a `CDataset`.
- #598 Added gradient method for `CKernelHistIntersection`.

Improved (6 changes)

- #562 Extended support of `CClassifierPyTorch` to nested PyTorch modules.
- #594 `CClassifierPyTorch.load_model()` is now able to also load models trained with PyTorch (without using our wrapper). New parameter `classes` added to the method to match classes to indexes in the loaded model.
- #579 Left side single row/column broadcast is now supported for sparse vs sparse `CArray` operations.
- #582 Improved performance of `CNormalizerMeanStd` when multiple channels are defined.
- #576 Vastly improved the performance of kernels by removing loops over samples in many classes and refactoring main routines.
- #562 Improved `grad_f_x` computation at a specific layer in `CClassifierPyTorch`.

Changed (4 changes)

- #578 `CClassifierPyTorch` now inherits from `CClassifierDNN`. The following changed accordingly: parameter `torch_model` renamed to `model`; property `layer_shapes` is now defined; method `save_checkpoint` removed.
- #562 Parameter `layer` of `CClassifierPyTorch.get_layer_output()` is now renamed `layer_names` as a list of layers names is supported (a single layer name is still supported as input). A dictionary is returned if multiple layers are requested. See the documentation for more information.
- #533 Double initialization in `CAttackEvasionPGDLS` will now be executed regardless of the classifier type (linear or nonlinear) if the `double_init` parameter of `.run()` method is set to `True`.
- #591 It is now not required to call the `fit` method of `CNormalizerMeanSTD` if fixed mean/std values are used.

Fixed (4 changes)

- #561 Fixed `CConstraintBox` not always applied correctly for float data.
- #577 Fixed `CClassifierPyTorch.decision_function` applying preprocess twice.
- #581 Fixed gradient computation of `CKernelChebyshevDistance`.
- #599 Kernels using distances are now based on negative distances (to correctly represent similarity measures). Affected classes are: `CKernelChebyshevDistance`, `CKernelEuclidean`.

Removed & Deprecated (5 changes)

- #561 Removed parameter `precision` from `CConstraint.is_violated()`.
- #575 Parameter `batch_size` of `CKernel` is now deprecated.
- #597 Removed unused parameter `gamma` from `CKernelChebyshevDistance`.
- #596 Removed `CKernelHamming`.
- #602 Renamed `CNormalizerMeanSTD` to `CNormalizerMeanStd`. The old class has been deprecated and will be removed in a future version.

Documentation (5 changes)

- #538 Added a notebook tutorial on the use of Explainable ML methods provided by the `secml.explanation` package.
- #573 Improved visualization of attack results in `07-ImageNet` tutorial.
- #610 Fixed spacing between parameter and parameter type in the docs.
- #605 Fixed documentation of classes requiring extra components not being displayed.
- #608 Added acknowledgments to `README`.

v0.9 (11/10/2019)

- #536 Added `CClassifierPytorch` to support Neural Networks (NNs) through PyTorch deep learning platform.

Improved (1 change)

- #556 `CFigure.imshow` now supports `PIL` images as input.

Changed (1 change)

- #532 Method `CPreProcess.revert()` is now renamed `.inverse_transform()`.

Fixed (1 change)

- #554 Fixed `CClassifierSkLearn.predict()` not working when using pretrained sklearn models.

Documentation (2 changes)

- #559 Deprecated functions and classes are now correctly visualized in the documentation.
- #560 Updated development roadmap accordingly to `0.10`, `0.11` and `0.12` releases.

Deprecations (3 changes)

- #532 Method `CPreProcess.revert()` is deprecated. Use `.inverse_transform()` instead.
- #552 `CClassifierKDE` is now deprecated. Use `CClassifierSkLearn` with `sklearn.neighbors.KernelDensity` instead.
- #553 `CClassifierMCSLinear` is now deprecated. Use `CClassifierSkLearn` with `sklearn.ensemble.BaggingClassifier` instead.

v0.8.1 (05/09/2019)

This version does not contain any significant change.

Documentation (2 changes)

- #523 Fixed documentation not compiling under Sphinx v2.2.
- #529 Updated roadmap accordingly for v0.9 release.

v0.8 (06/08/2019)

- First public release!

4.7.19 ROADMAP

SecML is still in alpha stage and the roadmap is subject to change at any time.

1. (Q1 2020) Support for Tensorflow 2 library
2. (Q2 2020) Foolbox library wrapper
3. (Q2 2020) Keras library wrapper

For further details and the most up-to-date roadmap see: https://gitlab.com/secml/secml/-/milestones

4.7.20 Contributing

Your contribution to the development of SecML is foundamental!

There are many ways to contribute to the SecML project. The most valuable contributions for us are bug reports, which help finding and fixing possible problems within the code. Feature requests are also a great way to provide feedback on the current development directions.

We also appreciate the contributions that extend our library by connecting it to most-used ML libraries and by adding state-of-the art attacks and defenses to use in experiments. Other useful contributions are documentation and examples of usage, which will greatly help us enlarge our user community. See Code Contributions and Extending SecML for more detailed information.

Finally, another way of contributing is sharing our work with colleagues and people that may be interested in using it for their experiments.

Submitting a bug report or feature request

Before creating an issue we kindly ask you to read the documentation to make sure the problem is not already covered. If not, please submit a *bug report* or a *feature request*.

Bug report

Please ensure the bug has not been already reported. If you're unable to find an open issue addressing a specific problem, open a new issue.

Be sure to include a clear title and description, as much relevant information as possible and, if applicable, a code sample or an executable test case demonstrating the expected behavior that is not occurring.

Feature request

Suggestions and feedback are always welcome. We ask to open an new issue for this purpose. Of course, you are free to contribute yourself to the code. See: Code Contributions.

4.7.21 Code Contributions

Contribution consisting on new code the library are much welcome. One of our maintainers will review the contributions and will help with the needed changes before integration, if any.

Development Installation

Start by creating a fork of our repository. Then, set up the project locally by the following means:

1. Install from local GitLab repository:
   - Clone the project repository in a directory of your choice
   - Run installation as: `pip install .`
2. Install from remote GitLab repository. In this case, given `{repourl}` in the format, es., `gitlab.com/secml/secml`:
   - `pip install git+ssh://git@{repourl}.git[@branch]#egg=secml` A specific branch to install can be specified using `[@branch]` parameter. If omitted, the default branch will be installed.

Contributions can be sent in the form of a merge request via our GitLab issue tracker. See how to create a merge request guide for more information.

Editable Installation

For SecML developers or users want to use the latest `dev` version of the library (soon available to the public), `pip` provides a convenient option which is called: **editable mode**.

By calling `pip install` with the `-e` option or `python setup.py develop`, only a reference to the project files is "installed" in the active environment. In this way, project files can be edited/updated and the new versions will be automatically executed by the Python interpreter.

Two common scenarios are listed below:

1. Editable install from a previously cloned local repository
    - Navigate to the repository directory
    - Run `python setup.py develop`
2. Editable install from remote repository
    - Run `pip install -e git+ssh://git@{repourl}.git[@branch]#egg=secml`
    - Project will be cloned automatically in `<venv path>/src/secml`
    - The new repository can then be updated using standard `git` commands

Merge request checklist

Please follow this checklist before sending a new merge request:

1. Use informative names for classes, methods, functions and variables.
2. Make sure your code passes the existing tests. Remember to test both CPU and GPU (CUDA) mode, if applicable.
3. Make sure your code is well documented and commented when possible. Make sure the documentation renders properly by compiling it.
4. Add tests if you are contributing to a new feature.
5. Make sure your code does not violate PEP-8 codestyle convention.
6. When applicable, re-use the code in the library without rewriting procedures that are already implemented somewhere else.
7. (optional) Provide an example of usage in the merge request, so that the contribution to the library will become clear to the reviewers as well as other contributors.

Coding guidelines

In this section, we summarize standards and conventions used in our library.

Code style

We follow python PEP-8 convention for ensuring code readability. 4-spaces indentation should be used.

Documentation style

We use informative docstrings for our code. Make sure your code is always commented and documented. The docstrings should follow the NumPy documentation format.

To locally build the documentation, run the following command: `tox -e docs`.

Compiled files will be available in the `docs/build/html` folder.

Packages

Our packages are nested inside macro-categories. Every package can contain modules, other packages or just directories for keeping everything structured and tidy.

Modules

We use separate files for each class so that they can be easily found within the package structure. Modules can also be created to group utility functions together.

Classes

Our class names all start with `C + <class_name>`, e.g. `CClassifier`. Hidden utility classes, accessible only internally from other classes, have names starting with underscores (`_C + <class_name>`).

The packages's superclass often expose public methods that call inner abstract methods. If you are subclassing one of these classes, take care of reading the superclass code and check out the inner methods that you need to implement. See: Extending SecML.

Tests

We test our code with pervasive unit tests, build on Python's unittest framework. Existing unittests can be run using tox.

You can also contribute to writing tests for our code. We have `tests` subdirectories in all our packages.

The main interface from which new tests should be inherited is the *`secml.testing.c_unittest.CUnitTest`* class.

Tests should run smoothly and fast, performing accurate initialization and cleanup. Implement the initialization in the `setUp` method, and the single test cases in other separate methods.

New test modules should be have name starting with `test_`.

4.7.22 Extending SecML

We provide details on how to implement new library modules, extending our abstract interfaces.

Remember to check out latest version and roadmap before developing new code.

Abstract Base Classes

The packages's abstract superclasses (e.g. *CClassifier*) expose public methods that call inner abstract methods. If you are creating a new extension, inherit from one of these classes, taking care of reading the superclass code and check out the inner methods that you need to implement.

New extensions should handle our main data type *CArray*. This class wraps the dense numpy `numpy.ndarray` and the scipy sparse `scipy.sparse.csr_matrix`, so that they have the same interface for the user.

The shape of a `CArray` is either a vector or a matrix (multi-dimensional arrays will be added in a future version) of rows where each row represents a sample.

Two `CArray` are needed to compose a *CDataset* that can be used to store samples (attribute X) and labels (attribute Y).

Creating new extensions

The following guides illustrate how to extend the superclasses for the different packages of the library:

CClassifier

The unified interface *CClassifier* defines the structure of classifiers.

We differentiate a standard classifier from Deep Neural Networks (DNNs), which require a more advanced interface defined by *CClassifierDNN* (described below).

Standard classifiers (`CClassifier`)

List of methods to implement:

- `_forward`: performs a forward pass of the input x. It should return the output of the decision function of the classifier.
- `_backward`: this method returns the gradient of the decision function output with respect to data. It takes a `CArray` as input, `w`, which pre-multiplies the gradient as in standard reverse-mode autodiff.
- `_fit`: fit the classifier on the data. Takes as input a `CDataset`.

DNN backends (`CClassifierDNN`)

The backend for DNN (*CClassifierDNN*) is based on the `CClassifier` interface, adding more methods specific to DNNs and their frameworks.

An example of how to extend the `CClassifierDNN` interface is our PyTorch wrapper *CClassifierPyTorch*.

List of methods to implement:

- `_forward`: performs a forward pass of the input x. It is slightly different from the `_forward` method of `CClassifier`, as it returns the output of the layer of the DNN specified in the attribute `_out_layer`. If `_out_layer` is `None`, the last layer output is returned (applies the softmax if `softmax_outputs` is True).
- `_backward`: returns the gradient of the output of the DNN layer specified in `_out_layer`, with respect to the input data.
- `_fit`: trains the classifier. Takes as input a `CDataset`.
- `layers` (property): returns a list of tuples containing the layers of the model, each tuple is structured as `(layer_name, layer)`.
- `layer_shapes` (property): returns the output shape of each layer (as a dictionary with layer names as keys).
- `_to_tensor`: converts a `CArray` into the tensor data type of the backend framework.
- `_from_tensor`: converts a backend tensor data type to a `CArray`.
- `save_model`: saves the model weight and parameters into a gz archive. If possible, it should allow model restoring as a checkpoint, i.e. the user should be able to continue training of the restored model.
- `load_model`: restores the model. If possible, it restores also the optimization parameters as the user may need to continue training.

It may be necessary to implement a custom data loader for the specific DNN backend. The data loader should take as input a `CDataset` and load the data for the backend. This is necessary because the inputs to the network may have their own shapes, whereas the `CArray` treats each sample as a row vector. We suggest to add the `input_shape` as an input parameter of the wrapper and handle the conversion inside.

- CClassifier - classifiers including Deep Neural Networks.

The following contribution guides will be added in a future versions.

- Data processing
  - `CPreprocess`
  - `CKernel`
- Data
  - `CDataLoader`
- Visualization
  - `CPlot`
- Attacks
  - *`CAttack`*
  - *`CAttackEvasion`*
  - *`CAttackPoisoning`*
- Optimization
  - *`COptimizer`*

BIBLIOGRAPHY

[1] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 67. http://www.math.sfu.ca/~cbm/aands/

[2] Wikipedia, “Logarithm”. http://en.wikipedia.org/wiki/Logarithm

[1] M. Abramowitz and I.A. Stegun, “Handbook of Mathematical Functions”, 10th printing, 1964, pp. 67. http://www.math.sfu.ca/~cbm/aands/

[2] Wikipedia, “Logarithm”. http://en.wikipedia.org/wiki/Logarithm

[1] G. Strang, *Linear Algebra and Its Applications*, 2nd Ed., Orlando, FL, Academic Press, Inc., 1980, pp. 139-142.

[1] “Lecture Notes on the Status of IEEE 754”, William Kahan, http://www.cs.berkeley.edu/~wkahan/ieee754status/IEEE754.PDF

[2] “How Futile are Mindless Assessments of Roundoff in Floating-Point Computation?”, William Kahan, http://www.cs.berkeley.edu/~wkahan/Mindless.pdf

[1] Spillmann, Barbara, et al. “Transforming strings to vector spaces using prototype selection.” Structural, Syntactic, and Statistical Pattern Recognition. Springer Berlin Heidelberg, 2006. 287-296.

[1] Rosenbrock, HoHo. “An automatic method for finding the greatest or least value of a function.” The Computer Journal 3.3 (1960): 175-184.

PYTHON MODULE INDEX

a

c

d

e

f

g

k

l

m

p

s

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z