# Conmo

***Release 1.0.1***

**Grupo de Metrología y Modelos (MyM)**

**Jun 23, 2022**

# CONTENTS

Release v1.0.1.

CHAPTER

ONE

# WHAT IS CONMO?

**Conmo** is a framework developed in Python whose main objective is to facilitate the execution and comparison of different experiments mainly related to Anomaly Detection and Condition Monitoring. These experiments consist of a series of concatenated stages forming a pipeline architecture, i.e. the output of one stage is the input of the next one. This framework aims to provide a way to standarize machine learning experiments, thus being able to reconstruct result tables of scientific papers.

CHAPTER

TWO

# USER GUIDE

## 2.1 Quickstart Guide

### 2.1.1 Requirements

**Conmo** was developed under Python version 3.7.11 so it should work with similar or more recent versions. However, we cannot claim this to be true, so we recommend using the same version. To be able to use **Conmo** you need to have installed a Python interpreter and the following libraries on your computer:

- Numpy
- Pandas
- Tensorflow
- Scikit-Learn
- Scipy
- Requests
- Pyarrow

If you want to make a contribution by modifying code and documentation you need to include these libraries as well:

- Sphinx
- Sphinx-rtd-theme
- Isort
- Autopep8

We suggest to create a new virtual enviroment using the Conda package manager and install there all dependences.

### 2.1.2 Installation

The fastest way to get started with Conmo is to install it via the pip command.

```
pip install conmo
```

And then you will be able to open a Python interpreter and try running.

```
import conmo
```

Some TensorFlow warnings might come up if your computer doesn't have installed a GPU, although that's not a problem for running Conmo.

You can also install Conmo manually downloading the source code from the Github repository.

```
git clone https://github.com/MyM-Uniovi/conmo.git
cd conmo
```

Then if you haven't prepared manually a conda enviroment, you can execute the shell-script `install_conmo_conda.sh` to install all the dependences and create a Conda enviroment with Python 3.7.

```
cd scripts
./install_conmo_conda.sh conda_env_name
```

If your operating system is not Unix-like and you are using Windows 10/11 OS you can create the Conda enviroment manually or use the Windows Subsytem for Linux (WSL) tool. For more information about its installation, please refer to Microsoft's official documentation..

To check if the Conda enviroment is activated you should see a `(conda_env_name)` in your command line. If it is not activated, then you can activated it using:

```
conda activate conda_env_name
```

### 2.1.3 Overview

The experiments in Conmo have a pipeline-based architecture. A pipeline consists of a chain of processes connected in such a way that the output of each element of the chain is the input of the next, thus creating a data flow. Each of these processes represents one of the typical generic steps in Machine Learning experiments:

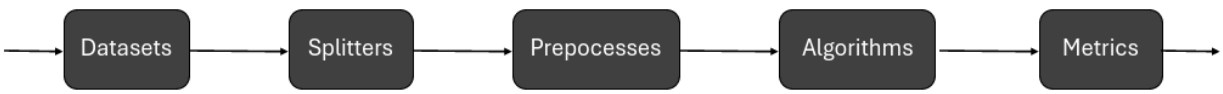
**Datasets**
: Defines the dataset used in the experiment which will be the starting data of the chain. Here the dataset will be loaded and parsed to a standard format.

**Splitters**
: Typically in Machine Learning problems the data has to be splitted into train data and test data. Also here you can apply Cross-Validation techniques.

**Preprocesses**
: Defines the sequence of preprocesses to be applied over the dataset to manipulate the data before any algorithm is executed.

**Algorithms**
: Defines the different algorithms which will be executed over the same input data stream (as a result of the previous stage). It can be one or several.

**Metrics**
: Defines the different metrics that can be used to evaluate the results obtained from the algorithms.




Further details and documentation about modules, functions and parameters are provided in the *API Reference*.

### 2.1.4 Running an experiment

Here is a brief example on how to use the different comno modules to reproduce an experiment. In this case with the predefined splitter of the Server Machine Dataset, Sklearn's MinMaxScaler as preprocessing, PCAMahalanobis as algorithm and accuracy as metric.

1. Import the module if it hasn't been imported yet and other dependences:

```
from sklearn.preprocessing import MinMaxScaler

from conmo import Experiment, Pipeline
from conmo.algorithms import PCAMahalanobis
from conmo.datasets import ServerMachineDataset
from conmo.metrics import Accuracy
from conmo.preprocesses import SklearnPreprocess
from conmo.splitters import SklearnSplitter
from sklearn.model_selection import PredefinedSplit
from sklearn.preprocessing import MinMaxScaler
```

2. Configure the different stages of the pipeline:

```
dataset = ServerMachineDataset('1-01')
splitter = SklearnSplitter(splitter=PredefinedSplit(dataset.sklearn_
↪predefined_split()))
preprocesses = [
    SklearnPreprocess(to_data=True, to_labels=False,
                      test_set=True, preprocess=MinMaxScaler()),
]
algorithms = [
    PCAMahalanobis()
]
metrics = [
    Accuracy()
]
pipeline = Pipeline(dataset, splitter, preprocesses, algorithms, metrics)
```

3. Create an experiment with the configured pipeline. The first parameter is a list of the pipelines that will be included in the experiment It can be one or more. The second parameter is for statistical testing between results, but this part is still under development and therefore it cannot be used:

```
experiment = Experiment([pipeline], [])
```

4. Start running the experiment by calling `launch()` method:

```
experiment.launch()
```

5. As a result of the execution of the experiment a specific folder structure will be created in `~/conmo`:

**`/data`**
: This directory contains the various datasets that have already been imported (downloaded and parsed) and are therefore already available for use. *They are stored in parquet format for better compression.* For each of the subdatasets included in each dataset, there will be a data file and a labels file.

**`/experiments`**
: This directory contains all the executions of an experiment in Conmo in chronological order. Each directory corresponds to an experiment and has in its name a timestamp with the time and day when this experiment was

run. Within each experiment directory there will be another one for each pipeline, and within this one there will be as many directories as the number of steps each pipeline has been determined to contain These folders contain the input and output data used by each step of the pipeline. They are also stored in parquet format, in the same way as the datasets in the `/data` folder.

CHAPTER

THREE

# EXAMPLES

## 3.1 Examples

A handful of example experiments can be found in the "examples" directory of the repository. These are listed below:

### 3.1.1 NASA TurboFan Degradation

This example can be found in *nasa_cmapss.py* file. The chosen dataset is NASA's Turbofan engine degradation simulation data set. It is a dataset widely used in multivariate time series anomaly detection and condition monitoring problems. The splitter used is the Sklearn Predefined Split. For more information see the Scikit-Learn documentation. Regarding preprocessing, several have been used. The Savitzky-Golay filter, RUL Imputation and Binarizer are already implemented in Conmo. The MinMaxScaler is a Sklearn preprocessing (more information here) that has been packaged using SklearnPreprocess. Finally, two custom preprocesses for data cleaning and label renaming have been defined using the CustomPreprocess wrapper. To create these preprocesses just create a function that has as parameters the Pandas Dataframes for data and labels. The algorithms used were dimensionality reduction with PCA together with Mahalanobis distance calculation and One Class Support Vector Machine. Finally, the metric used was Acurracy.

```
import pandas as pd
from sklearn.model_selection import PredefinedSplit
from sklearn.preprocessing import MinMaxScaler

from conmo import Experiment, Pipeline
from conmo.algorithms import OneClassSVM, PCAMahalanobis
from conmo.datasets import NASATurbofanDegradation
from conmo.metrics import Accuracy
from conmo.preprocesses import (Binarizer, CustomPreprocess, RULImputation,
                                SavitzkyGolayFilter, SklearnPreprocess)
from conmo.splitters import SklearnSplitter

# First custom preprocess definition
def data_cleanup(data: pd.DataFrame, labels: pd.DataFrame) -> (pd.DataFrame, pd.
↪DataFrame):
    # Reduce columns
    columns = ['T30', 'T50', 'P30']
    sub_data = data.loc[:, columns]

    # Rename columns
    sub_data = sub_data.rename(columns={'T50': 'TGT'})

    # Calculate FF
```

(continues on next page)

(continued from previous page)

```
    sub_data.loc[:, 'FF'] = data.loc[:, 'Ps30'] * data.loc[:, 'phi']
    sub_data.head()

    return sub_data, labels

# Second custom preprocess definition
def rename_labels(data: pd.DataFrame, labels: pd.DataFrame) -> (pd.DataFrame, pd.
↪DataFrame):
    # Rename labels from 'rul' to 'anomaly'
    labels.rename(columns={'rul': 'anomaly'}, inplace=True)

    return data, labels


# Select FD001 subdataset of NASA Turbofan Degradation dataset
dataset = NASATurbofanDegradation(subdataset="FD001")

# Split dataset using predefined dataset split
splitter = SklearnSplitter(splitter=PredefinedSplit(dataset.sklearn_predefined_split()))

# Preprocesses definition
preprocesses = [
    CustomPreprocess(data_cleanup),
    SklearnPreprocess(to_data=True, to_labels=False,
                      test_set=True, preprocess=MinMaxScaler()),
    SavitzkyGolayFilter(to_data=True, to_labels=False,
                          test_set=True, window_length=7, polyorder=2),
    RULImputation(threshold=125),
    Binarizer(to_data=False, to_labels=[
                      'rul'], test_set=True, threshold=50),
    CustomPreprocess(rename_labels)
]

# Algorithms definiition with default parameters
algorithms = [
    PCAMahalanobis(),
    OneClassSVM()
]

metrics = [
    Accuracy()
]
# Pipeline with all steps
pipeline = Pipeline(dataset, splitter, preprocesses, algorithms, metrics)

# Experiment definition and launch
experiment = Experiment([pipeline], [])
experiment.launch()
```

### 3.1.2 Batteries Degradation

This experiment can be found in the file *batteries_degradation.py* and reproduces the results obtained in a paper to estimate the level of degradation of some types of lithium batteries. The dataset used is Batteries Degradation. This is not a time series, although it is somewhat similar since it measures different types of degradation in three types of batteries as they are gradually used. It is a local dataset, so it is necessary to pass the path in which it is located, and also the type of battery to be selected (LFP) and the test set, in this case 1. The splitter used is the Sklearn Predefined Split and it does not have any preprocessing since during the parsing of the local files to the Conmo format the data is already normalised. The algorithms used are the same as those used in the paper: Random Forest, Multilayer Perceptron and Convolutional Neural Network. In all cases the pre-trained models are used, so it is necessary to pass the path to the files as a parameter. The metric used is Root Mean Square Percentage Error.

```
from conmo import Experiment, Pipeline
from conmo.algorithms import PretrainedRandomForest, PretrainedCNN1D,␣
↪PretrainedMultilayerPerceptron
from conmo.datasets import BatteriesDataset
from conmo.metrics import RMSPE
from conmo.splitters import SklearnSplitter
from sklearn.model_selection import PredefinedSplit

# Pipeline definition
# Change path to our local dataset files, specify chemistry of the batteries (LFP, NCA,␣
↪NMC) and test set
dataset = BatteriesDataset('/path/to/batteries/dataset/', 'LFP', 1)
splitter = SklearnSplitter(splitter=PredefinedSplit(dataset.sklearn_predefined_split()))
preprocesses = None
# Changes the path to the files where the pre-trained models are stored (usually h5,␣
↪h5py or joblib formats).
algorithms = [
    PretrainedRandomForest(pretrained=True, path='/path/to/saved/model-RF.joblib'),
    PretrainedMultilayerPerceptron(pretrained=True, input_len=128, path='/path/to/saved/
↪model-MLP.h5'),
    PretrainedCNN1D(pretrained=True, input_len=128, path='/path/to/saved/model-CNN.h5')
]
metrics = [
    RMSPE()
]
pipeline = Pipeline(dataset, splitter, preprocesses, algorithms, metrics)


# Experiment definition and launch
experiment = Experiment([pipeline], [])
experiment.launch()
```

### 3.1.3 Server Machine Dataset with PCAMahalanobis

This experiment can be found in the file *omni_anomaly_smd.py*. The Server Machine Dataset used in this experiment has been obtained from the OmniAnomaly repository. In their Github you can find more information about the dataset as well as the implementation of other anomaly detection and time series data mining algorithms. The splitter used is the Sklearn Predefined Split and the preprocessing is the MinMaxScaler from Sklearn. The algorithms is PCA with Mahalanobis distance. Finally, the metric is the Accuracy.

```
from sklearn.preprocessing import MinMaxScaler

from conmo import Experiment, Pipeline
from conmo.algorithms import PCAMahalanobis
from conmo.datasets import ServerMachineDataset
from conmo.metrics import Accuracy
from conmo.preprocesses import SklearnPreprocess
from conmo.splitters import SklearnSplitter
from sklearn.model_selection import PredefinedSplit
from sklearn.preprocessing import MinMaxScaler

# Pipeline definition
dataset = ServerMachineDataset('1-01')
splitter = SklearnSplitter(splitter=PredefinedSplit(dataset.sklearn_predefined_split()))
preprocesses = [
    SklearnPreprocess(to_data=True, to_labels=False,
                      test_set=True, preprocess=MinMaxScaler()),
]
algorithms = [
    PCAMahalanobis()
]
metrics = [
    Accuracy()
]
pipeline = Pipeline(dataset, splitter, preprocesses, algorithms, metrics)


# Experiment definition and launch
experiment = Experiment([pipeline], [])
experiment.launch()
```

CHAPTER

FOUR

# API REFERENCE

## 4.1 API Reference

This is the API Reference documentation of the package, including modules, classes and functions.

### 4.1.1 conmo.experiments

This is the main submodule of the package and it is the responsible of create the intermediary directories of the experiment and take care of creating and executing the configured pipeline.

| | |
|---|---|
| *experiment.Experiment*(pipelines, analytics) | |
| *experiment.Pipeline*(dataset, splitter, ...) | |

#### conmo.experiment.Experiment

**class** conmo.experiment.**Experiment**(*pipelines: Iterable[*Pipeline*]*, *analytics: Iterable*, *name='2022_06_23-10_04_44'*)

**__init__**(*pipelines: Iterable[*Pipeline*]*, *analytics: Iterable*, *name='2022_06_23-10_04_44'*)

**generate_dirs**() → Iterable[str]

Generates directories both for the experiment itself and for the pipelines it contains.

**Returns**

**pipes_dirs** – Array containing the names of the directories of the pipes of the current experiment.

**Return type**

Iterable[str]

**launch**()

Launchs the current experiment.

### Methods

| | |
|---|---|
| *__init__*(pipelines, analytics[, name]) | |

| | |
|---|---|
| *generate_dirs*() | Generates directories both for the experiment itself and for the pipelines it contains. |
| *launch*() | Launchs the current experiment. |

## conmo.experiment.Pipeline

**class** conmo.experiment.**Pipeline**(*dataset: Dataset*, *splitter: Optional[Splitter]*, *preprocesses: Optional[Iterable[Preprocess]]*, *algorithms: Iterable[Algorithm]*, *metrics: Iterable[Metric]*)

**__init__**(*dataset: Dataset*, *splitter: Optional[Splitter]*, *preprocesses: Optional[Iterable[Preprocess]]*, *algorithms: Iterable[Algorithm]*, *metrics: Iterable[Metric]*) → None

**generate_dirs**(*pipe_dir: str*) → None

Auxiliary method to generate directories for each of the steps in the current pipeline.

**Parameters**

**pipe_dir** (*str*) – Name of the pipe directory.

**run**(*pipe_dir: str*, *pipe_num: int*, *pipes: int*) → None

Contains all the logic for the execution of a particular pipeline, creating intermediate directories for data passing and executing the relevant methods for each step.

**Parameters**

- **pipe_dir** (*str*) – Name of the current pipeline directory.
- **pipe_num** (*int*) – Index of the current pipelines.
- **pipes** (*int*) – Total number of pipelines in the current experiment.

### Methods

| | |
|---|---|
| *__init__*(dataset, splitter, preprocesses, ...) | |

| | |
|---|---|
| *generate_dirs*(pipe_dir) | Auxiliary method to generate directories for each of the steps in the current pipeline. |
| *run*(pipe_dir, pipe_num, pipes) | Contains all the logic for the execution of a particular pipeline, creating intermediate directories for data passing and executing the relevant methods for each step. |

#### Attributes

| |
|---|
| ALGORITHMS_FOLDER |
| DATASET_FOLDER |
| METRICS_FOLDER |
| PREPROCESSES_FOLDER |
| SPLITTER_FOLDER |

### 4.1.2 conmo.datasets

The *conmo.datasets* submodule takes care of downloading the dataset and parsing it to the Conmo's format.

| | |
|---|---|
| *datasets.dataset.Dataset*(name) | Abstract base class for a Dataset. |
| *datasets.dataset.RemoteDataset*(url, ...) | Abstract base class for a RemoteDataset (downloadable). |
| *datasets.dataset.LocalDataset*(path) | Abstract base class for a LocalDataset (loadable). |
| *datasets.MarsScienceLaboratoryMission*(channel) | |
| *datasets.SoilMoistureActivePassiveSatellite*(channel) | |
| *datasets.ServerMachineDataset*(subdataset) | |
| *datasets.NASATurbofanDegradation*(subdataset) | |
| *datasets.BatteriesDataset*(path, chemistry, ...) | This is a dataset obtained from measurements of certain types of degradation of three types of batteries. Since it belongs to the local datasets, to launch any experiment with it, it must be stored on disk with the following directory structure: - DTW-Li-ion-Diagnosis - data : Data and labels for the three types of batteries are stored here. - mat: - LFP: - diagnosis: - V.mat - test: - V_references.mat - x_test_0.mat - x_test_1.mat - x_test_2.mat - x_test_3.mat - y_test.mat - NCA: - diagnosis - test - NMC: - The same as NCA and LFP - Q.mat. |

#### conmo.datasets.dataset.Dataset

**class** conmo.datasets.dataset.**Dataset**(*name: str*)

Abstract base class for a Dataset.

This class is an abstract class from which other subclasses inherit and must not be instanciated directly.

**__init__**(*name: str*) → None

Main constructor of the class.

**Parameters**

**name** (*str*) – The name given to the dataset.

**abstract dataset_files**() → Iterable

Iterable of files included in the dataset.

**abstract fetch**(*out_dir: str*) → None

Fetch data to feed the pipeline.

**Parameters**

**out_dir** (*str*) – Directory where the dataset will be stored.

**is_dataset_ready**() → bool

Check if dataset has been already loaded/downloaded and parsed to package format.

**show_start_message**() → None

Show starting step info message.

### Methods

| | |
|---|---|
| *__init__*(name) | Main constructor of the class. |
| *dataset_files*() | Iterable of files included in the dataset. |
| *fetch*(out_dir) | Fetch data to feed the pipeline. |
| *is_dataset_ready*() | Check if dataset has been already loaded/downloaded and parsed to package format. |
| *show_start_message*() | Show starting step info message. |

## conmo.datasets.dataset.RemoteDataset

**class** conmo.datasets.dataset.**RemoteDataset**(*url: str*, *file_format: str*, *checksum: str*, *checksum_format: str*)

Abstract base class for a RemoteDataset (downloadable).

**__init__**(*url: str*, *file_format: str*, *checksum: str*, *checksum_format: str*) → None

Main constructor of the class.

**Parameters**

**name** (*str*) – The name given to the dataset.

**check_checksum**(*response: object*) → bool

Checks if the checksum of the downloaded file corresponds to the one provided in the class. For security e integrity issues. Currently only the md5 algorithm is integrated.

**Parameters**

**response** (*Object*) – Response object returned by the get method of the Requests library.

**Return type**

Boolean variable indicating whether the comparison of the hash with the checksum was successful or not.

**abstract dataset_files**() → Iterable

Iterable of files included in the dataset.

**download**(*out_dir: str*) → None

Download a Dataset from a remote URL.

**extract_data**(*response: object*, *out_dir: str*) → None

Extracts the contents of a compressed file in zip format.

**Parameters**

- **response** (*Object*) – Response object returned by the get method of the Requests library.
- **out_dir** (*str*) – Directory were the zip file will be unzziped.

**abstract feed_pipeline**(*out_dir: str*) → None

Copy selected data file to pipeline step folder.

**fetch**(*out_dir: str*) → None

Fetch data to feed the pipeline.

**Parameters**

**out_dir** (*str*) – Directory where the dataset will be stored.

**is_dataset_ready**() → bool

Check if dataset has been already loaded/downloaded and parsed to package format.

**abstract parse_to_package**(*raw_dir: str*) → None

Parse raw dataset to package format. Data and labels must be saved in parquet format. More information about parquet format: https://parquet.apache.org/

**Parameters**

**raw_dir** – Directory where the dataset was downloaded from its source.

**show_start_message**() → None

Show starting step info message.

### Methods

| | |
|---|---|
| *__init__*(url, file_format, checksum, ...) | Main constructor of the class. |
| *check_checksum*(response) | Checks if the checksum of the downloaded file corresponds to the one provided in the class. |
| *dataset_files*() | Iterable of files included in the dataset. |
| *download*(out_dir) | Download a Dataset from a remote URL. |
| *extract_data*(response, out_dir) | Extracts the contents of a compressed file in zip format. |
| *feed_pipeline*(out_dir) | Copy selected data file to pipeline step folder. |
| *fetch*(out_dir) | Fetch data to feed the pipeline. |
| *is_dataset_ready*() | Check if dataset has been already loaded/downloaded and parsed to package format. |
| *parse_to_package*(raw_dir) | Parse raw dataset to package format. |
| *show_start_message*() | Show starting step info message. |

## conmo.datasets.dataset.LocalDataset

**class** conmo.datasets.dataset.**LocalDataset**(*path: str*)

Abstract base class for a LocalDataset (loadable).

**__init__**(*path: str*) → None

Constructor of local dataset.

**Parameters**

**path** (*str*) – Absolute path to the folder where the dataset is located in your disk. See the example of batteries_degradation.py.

**abstract dataset_files**() → Iterable

Iterable of files included in the dataset.

**abstract feed_pipeline**(*out_dir: str*) → None

Copy selected data file to pipeline step folder.

**Parameters**

**out_dir** – Directory where the dataset was originally stored.

**fetch**(*out_dir: str*) → None

Fetch data to feed the pipeline.

**Parameters**

**out_dir** (*str*) – Directory where the dataset will be stored.

**is_dataset_ready**() → bool

Check if dataset has been already loaded/downloaded and parsed to package format.

**abstract load**() → None

Parse raw dataset to package format. Data and labels must be saved in parquet format. More information about parquet format: https://parquet.apache.org/

**show_start_message**() → None

Show starting step info message.

### Methods

| | |
|---|---|
| *__init__*(path) | Constructor of local dataset. |
| *dataset_files*() | Iterable of files included in the dataset. |
| *feed_pipeline*(out_dir) | Copy selected data file to pipeline step folder. |
| *fetch*(out_dir) | Fetch data to feed the pipeline. |
| *is_dataset_ready*() | Check if dataset has been already loaded/downloaded and parsed to package format. |
| *load*() | Parse raw dataset to package format. |
| *show_start_message*() | Show starting step info message. |

## conmo.datasets.MarsScienceLaboratoryMission

**class** conmo.datasets.**MarsScienceLaboratoryMission**(*channel: str*)

**__init__**(*channel: str*) → None

Main constructor of the class.

**Parameters**

**name** (*str*) – The name given to the dataset.

**check_checksum**(*response: object*) → bool

Checks if the checksum of the downloaded file corresponds to the one provided in the class. For security e integrity issues. Currently only the md5 algorithm is integrated.

**Parameters**

**response** (*Object*) – Response object returned by the get method of the Requests library.

**Return type**

Boolean variable indicating whether the comparison of the hash with the checksum was successful or not.

**check_checksum_lbl**(*response: object*, *checksum: str*) → bool

Checks if the checksum of the downloaded file corresponds to the one provided in the class. For security e integrity issues. Currently only the md5 algorithm is integrated. Since in the MLS

dataset the labels are obtained from a different file, it's necessary to use another method to pass the checksum of that file.

**Parameters**

- **response** (*Object*) – Response object returned by the get method of the Requests library.
- **checksum** (*str*) – String containing the labels checksum.

**Returns**

Boolean variable indicating whether the comparison of the hash with the checksum was successful or not.

**Return type**

bool

**dataset_files**() → Iterable

Iterable of files included in the dataset.

**download**(*out_dir: str*) → None

Download a Dataset from a remote URL.

**download_anomalies_file**(*raw_dir: str*) → Iterable[DataFrame]

Method in charge of downloading and parsing the MLS dataset labels files. This is because the tags are located at a different URL than the data.

**Parameters**

**raw_dir** (*str*) – Directory were the unparsed data of SMAP dataset is stored until it's processed.

**Returns**

**labeled_anomalies** – Anomalous intervals in the MSL dataset.

**Return type**

Pandas Dataframe

**extract_data**(*response: object*, *out_dir: str*) → None

Extracts the contents of a compressed file in zip format.

**Parameters**

- **response** (*Object*) – Response object returned by the get method of the Requests library.
- **out_dir** (*str*) – Directory were the zip file will be unzziped.

**feed_pipeline**(*out_dir: str*) → None

Copy selected data file to pipeline step folder.

**fetch**(*out_dir: str*) → None

Fetch data to feed the pipeline.

**Parameters**

**out_dir** (*str*) – Directory where the dataset will be stored.

**is_dataset_ready**() → bool

Check if dataset has been already loaded/downloaded and parsed to package format.

**parse_to_package**(*raw_dir: str*) → None

Parse raw dataset to package format. Data and labels must be saved in parquet format. More information about parquet format: https://parquet.apache.org/

**Parameters**

**raw_dir** – Directory where the dataset was downloaded from its source.

**represent_anomalies**(*labels: Iterable[DataFrame]*, *channel: str*, *labeled_anomalies: Iterable[DataFrame]*) → Iterable[DataFrame]

Represent anomalies in the label's dataset following the anomalous intervals of 'labeled_anomalies.csv'

**Parameters**
- **labels** (*Pandas Dataframe*) – Dataframe with the shape of the labels but filled wth zeros.
- **channel** (*str*) – Channel identifier (subdataset).
- **labeled_anormalies** (*Pandas Dataframe*) – Anomalous intervals in the MSL dataset.

**Returns**
**labels** – Labels dataset correctly filled.

**Return type**
Pandas Dataframe

**show_start_message**() → None

Show starting step info message.

**sklearn_predefined_split**() → Iterable[int]

Generates array of indexes of same length as sequences to be used with 'PredefinedSplit' MSL dataset has only 2 sequences: one for train and another for test.

**Returns**
List with the index for each sequence of the dataset.

**Return type**
array

### Methods

| | |
|---|---|
| *__init__*(channel) | Main constructor of the class. |
| *check_checksum*(response) | Checks if the checksum of the downloaded file corresponds to the one provided in the class. |
| *check_checksum_lbl*(response, checksum) | Checks if the checksum of the downloaded file corresponds to the one provided in the class. |
| *dataset_files*() | Iterable of files included in the dataset. |
| *download*(out_dir) | Download a Dataset from a remote URL. |
| *download_anomalies_file*(raw_dir) | Method in charge of downloading and parsing the MLS dataset labels files. |
| *extract_data*(response, out_dir) | Extracts the contents of a compressed file in zip format. |
| *feed_pipeline*(out_dir) | Copy selected data file to pipeline step folder. |
| *fetch*(out_dir) | Fetch data to feed the pipeline. |
| *is_dataset_ready*() | Check if dataset has been already loaded/downloaded and parsed to package format. |
| *parse_to_package*(raw_dir) | Parse raw dataset to package format. |
| *represent_anomalies*(labels, channel, ...) | Represent anomalies in the label's dataset following the anomalous intervals of 'labeled_anomalies.csv' |
| *show_start_message*() | Show starting step info message. |
| *sklearn_predefined_split*() | Generates array of indexes of same length as sequences to be used with 'PredefinedSplit' MSL dataset has only 2 sequences: one for train and another for test. |

### Attributes

| |
|---|
| CHANNELS |
| CHECKSUM |
| CHECKSUM_FORMAT |
| FILE_FORMAT |
| LABEL |
| SEQUENCE_COLUMN |
| TIME_COLUMN |
| URL |
| VARIABLES |

## conmo.datasets.SoilMoistureActivePassiveSatellite

**class** conmo.datasets.**SoilMoistureActivePassiveSatellite**(*channel: str*)

**__init__**(*channel: str*) → None

Main constructor of the class.

**Parameters**

**name** (*str*) – The name given to the dataset.

**check_checksum**(*response: object*) → bool

Checks if the checksum of the downloaded file corresponds to the one provided in the class. For security e integrity issues. Currently only the md5 algorithm is integrated.

**Parameters**

**response** (*Object*) – Response object returned by the get method of the Requests library.

**Return type**

Boolean variable indicating whether the comparison of the hash with the checksum was successful or not.

**check_checksum_lbl**(*response: object*, *checksum: str*) → bool

Checks if the checksum of the downloaded file corresponds to the one provided in the class. For security e integrity issues. Currently only the md5 algorithm is integrated. Since in the SMAP dataset the labels are obtained from a different file, it's necessary to use another method to pass the checksum of that file.

**Parameters**

- **response** (*object*) – Response object returned by the get method of the Requests library.
- **checksum** (*str*) – String containing the labels' checksum.

**Returns**

Boolean variable indicating whether the comparison of the hash with the checksum was successful or not.

**Return type**
bool

**dataset_files**() → Iterable

Iterable of files included in the dataset.

**download**(*out_dir: str*) → None

Download a Dataset from a remote URL.

**download_anomalies_file**(*raw_dir: str*) → Iterable[DataFrame]

Method in charge of downloading and parsing the SMAP dataset labels files. This is because the tags are located at a different URL than the data.

**Parameters**
**raw_dir** (*str*) – Directory were the unparsed data of SMAP dataset is stored until it's processed.

**Returns**
**labeled_anomalies** – Anomalous intervals in the SMAP dataset.

**Return type**
Pandas Dataframe

**extract_data**(*response: object*, *out_dir: str*) → None

Extracts the contents of a compressed file in zip format.

**Parameters**
- **response** (*Object*) – Response object returned by the get method of the Requests library.
- **out_dir** (*str*) – Directory were the zip file will be unzziped.

**feed_pipeline**(*out_dir: str*) → None

Copy selected data file to pipeline step folder.

**fetch**(*out_dir: str*) → None

Fetch data to feed the pipeline.

**Parameters**
**out_dir** (*str*) – Directory where the dataset will be stored.

**is_dataset_ready**() → bool

Check if dataset has been already loaded/downloaded and parsed to package format.

**parse_to_package**(*raw_dir: str*) → None

Parse raw dataset to package format. Data and labels must be saved in parquet format. More information about parquet format: https://parquet.apache.org/

**Parameters**
**raw_dir** – Directory where the dataset was downloaded from its source.

**represent_anomalies**(*labels: Iterable[DataFrame]*, *channel: str*, *labeled_anomalies: Iterable[DataFrame]*) → Iterable[DataFrame]

Represent anomalies in the label's dataset following the anomalous intervals of 'labeled_anomalies.csv'

**Parameters**
- **labels** (*Pandas Dataframe*) – Dataframe with the shape of the labels but filled wth zeros.
- **channel** (*str*) – Channel identifier (subdataset)
- **labeled_anormalies** (*Pandas Dataframe*) – Anomalous intervals in the SMAP dataset.

**Returns**
**labels** – Labels dataset correctly filled.

**Return type**
Pandas Dataframe

**show_start_message**() → None

Show starting step info message.

**sklearn_predefined_split**() → Iterable[int]

Generates array of indexes of same length as sequences to be used with 'PredefinedSplit' SMAP dataset has only 2 sequences: one for train and another for test.

**Returns**
List with the index for each sequence of the dataset.

**Return type**
array

### Methods

| | |
|---|---|
| *__init__*(channel) | Main constructor of the class. |
| *check_checksum*(response) | Checks if the checksum of the downloaded file corresponds to the one provided in the class. |
| *check_checksum_lbl*(response, checksum) | Checks if the checksum of the downloaded file corresponds to the one provided in the class. |
| *dataset_files*() | Iterable of files included in the dataset. |
| *download*(out_dir) | Download a Dataset from a remote URL. |
| *download_anomalies_file*(raw_dir) | Method in charge of downloading and parsing the SMAP dataset labels files. |
| *extract_data*(response, out_dir) | Extracts the contents of a compressed file in zip format. |
| *feed_pipeline*(out_dir) | Copy selected data file to pipeline step folder. |
| *fetch*(out_dir) | Fetch data to feed the pipeline. |
| *is_dataset_ready*() | Check if dataset has been already loaded/downloaded and parsed to package format. |
| *parse_to_package*(raw_dir) | Parse raw dataset to package format. |
| *represent_anomalies*(labels, channel, ...) | Represent anomalies in the label's dataset following the anomalous intervals of 'labeled_anomalies.csv' |
| *show_start_message*() | Show starting step info message. |
| *sklearn_predefined_split*() | Generates array of indexes of same length as sequences to be used with 'PredefinedSplit' SMAP dataset has only 2 sequences: one for train and another for test. |

### Attributes

| |
|---|
| CHANNELS |
| CHECKSUM |
| CHECKSUM_FORMAT |
| FILE_FORMAT |
| LABEL |
| SEQUENCE_COLUMN |
| TIME_COLUMN |
| URL |
| VARIABLES |

## conmo.datasets.ServerMachineDataset

**class** conmo.datasets.**ServerMachineDataset**(*subdataset: str*)

**__init__**(*subdataset: str*) → None

Main constructor of the class.

**Parameters**

**name** (*str*) – The name given to the dataset.

**check_checksum**(*response: object*) → bool

Checks if the checksum of the downloaded file corresponds to the one provided in the class. For security e integrity issues. Currently only the md5 algorithm is integrated.

**Parameters**

**response** (*Object*) – Response object returned by the get method of the Requests library.

**Return type**

Boolean variable indicating whether the comparison of the hash with the checksum was successful or not.

**dataset_files**() → Iterable

Iterable of files included in the dataset.

**download**(*out_dir: str*) → None

Download a Dataset from a remote URL.

**extract_data**(*response: object*, *out_dir: str*) → None

Extracts the contents of a compressed file in zip format.

**Parameters**

- **response** (*Object*) – Response object returned by the get method of the Requests library.
- **out_dir** (*str*) – Directory were the zip file will be unzziped.

**feed_pipeline**(*out_dir: str*) → None

Copy selected data file to pipeline step folder.

**fetch**(*out_dir: str*) → None

Fetch data to feed the pipeline.

**Parameters**

**out_dir** (*str*) – Directory where the dataset will be stored.

**is_dataset_ready**() → bool

Check if dataset has been already loaded/downloaded and parsed to package format.

**parse_to_package**(*raw_dir: str*) → None

Parse raw dataset to package format. Data and labels must be saved in parquet format. More information about parquet format: https://parquet.apache.org/

**Parameters**

**raw_dir** – Directory where the dataset was downloaded from its source.

**show_start_message**() → None

Show starting step info message.

**sklearn_predefined_split**() → Iterable[int]

Generates array of indexes of same length as sequences to be used with 'PredefinedSplit' SMD dataset has only 2 sequences: one for train and another for test.

**Returns**

List with the index for each sequence of the dataset.

**Return type**

array

### Methods

| | |
|---|---|
| *__init__*(subdataset) | Main constructor of the class. |
| *check_checksum*(response) | Checks if the checksum of the downloaded file corresponds to the one provided in the class. |
| *dataset_files*() | Iterable of files included in the dataset. |
| *download*(out_dir) | Download a Dataset from a remote URL. |
| *extract_data*(response, out_dir) | Extracts the contents of a compressed file in zip format. |
| *feed_pipeline*(out_dir) | Copy selected data file to pipeline step folder. |
| *fetch*(out_dir) | Fetch data to feed the pipeline. |
| *is_dataset_ready*() | Check if dataset has been already loaded/downloaded and parsed to package format. |
| *parse_to_package*(raw_dir) | Parse raw dataset to package format. |
| *show_start_message*() | Show starting step info message. |
| *sklearn_predefined_split*() | Generates array of indexes of same length as sequences to be used with 'PredefinedSplit' SMD dataset has only 2 sequences: one for train and another for test. |

### Attributes

| |
|---|
| CHECKSUM |
| CHECKSUM_FORMAT |
| FILE_FORMAT |
| LABEL |
| SEQUENCE_COLUMN |
| SUBDATASETS |
| TIME_COLUMN |
| URL |
| VARIABLES |

## conmo.datasets.NASATurbofanDegradation

**class** conmo.datasets.**NASATurbofanDegradation**(*subdataset: str*)

**__init__**(*subdataset: str*) → None

Main constructor of the class.

**Parameters**

**name** (*str*) – The name given to the dataset.

**check_checksum**(*response: object*) → bool

Checks if the checksum of the downloaded file corresponds to the one provided in the class. For security e integrity issues. Currently only the md5 algorithm is integrated.

**Parameters**

**response** (*Object*) – Response object returned by the get method of the Requests library.

**Return type**

Boolean variable indicating whether the comparison of the hash with the checksum was successful or not.

**dataset_files**() → Iterable

Iterable of files included in the dataset.

**download**(*out_dir: str*) → None

Download a Dataset from a remote URL.

**extract_data**(*response: object*, *out_dir: str*) → None

Extracts the contents of a compressed file in zip format.

**Parameters**

- **response** (*Object*) – Response object returned by the get method of the Requests library.
- **out_dir** (*str*) – Directory were the zip file will be unzziped.

**feed_pipeline**(*out_dir: str*) → None

Copy selected data file to pipeline step folder.

**fetch**(*out_dir: str*) → None

Fetch data to feed the pipeline.

**Parameters**

**out_dir** (*str*) – Directory where the dataset will be stored.

**is_dataset_ready**() → bool

Check if dataset has been already loaded/downloaded and parsed to package format.

**parse_to_package**(*raw_dir: str*) → None

Parse raw dataset to package format. Data and labels must be saved in parquet format. More information about parquet format: https://parquet.apache.org/

**Parameters**

**raw_dir** – Directory where the dataset was downloaded from its source.

**show_start_message**() → None

Show starting step info message.

**sklearn_predefined_split**() → Iterable[int]

Generates array of indexes of same length as sequences to be used with 'PredefinedSplit'

**Returns**

List with the index for each sequence of the dataset.

**Return type**

array

### Methods

| | |
|---|---|
| *__init__*(subdataset) | Main constructor of the class. |
| *check_checksum*(response) | Checks if the checksum of the downloaded file corresponds to the one provided in the class. |
| *dataset_files*() | Iterable of files included in the dataset. |
| *download*(out_dir) | Download a Dataset from a remote URL. |
| *extract_data*(response, out_dir) | Extracts the contents of a compressed file in zip format. |
| *feed_pipeline*(out_dir) | Copy selected data file to pipeline step folder. |
| *fetch*(out_dir) | Fetch data to feed the pipeline. |
| *is_dataset_ready*() | Check if dataset has been already loaded/downloaded and parsed to package format. |
| *parse_to_package*(raw_dir) | Parse raw dataset to package format. |
| *show_start_message*() | Show starting step info message. |
| *sklearn_predefined_split*() | Generates array of indexes of same length as sequences to be used with 'PredefinedSplit' |

### Attributes

| |
|---|
| CHECKSUM |
| CHECKSUM_FORMAT |
| FILE_FORMAT |
| LABEL |
| SEQUENCE_COLUMN |
| SUBDATASETS |
| TIME_COLUMN |
| URL |
| VARIABLES |

## conmo.datasets.BatteriesDataset

**class** conmo.datasets.**BatteriesDataset**(*path: str*, *chemistry: str*, *test_set: int*)

This is a dataset obtained from measurements of certain types of degradation of three types of batteries. Since it belongs to the local datasets, to launch any experiment with it, it must be stored on disk with the following directory structure: - DTW-Li-ion-Diagnosis

- **data**
  [Data and labels for the three types of batteries] are stored here.
- **mat:**
  - **LFP:**
    - **diagnosis:**
      - V.mat
    - **test:**
      - V_references.mat
      - x_test_0.mat
      - x_test_1.mat
      - x_test_2.mat
      - x_test_3.mat
      - y_test.mat
  - **NCA:**
    - diagnosis
    - test
  - **NMC:**

        * The same as NCA and LFP

    - Q.mat

**IC**(*u: ~numpy.ndarray*, *q: ~numpy.ndarray*, *ui_step: float = 0.0005*, *minV: float = 3.2*, *maxV: float = 3.5) -> (<class 'numpy.ndarray'>, <class 'numpy.ndarray'>)*

Get the ICA data for a given voltage curve

**Parameters**

- **u** (*numpy array*) – Voltage curve.
- **q** (*numpy array*) – Capacity curve.
- **ui_step** (*float*) – Step of interpolation.
- **minV** (*float*) – Minimum voltage of the IC curve.
- **maxV** (*float*) – Maximum voltage of the IC curve.

**Returns**

**ui, dqi** – Interpolated voltage and derivative of capacity

**Return type**

numpy arrays

**__init__**(*path: str*, *chemistry: str*, *test_set: int*) → None

Constructor of local dataset.

**Parameters**

**path** (*str*) – Absolute path to the folder where the dataset is located in your disk. See the example of batteries_degradation.py.

**convert_to_input_data**(*ui_new: list*, *Q: list*, *size: int*, *material: int*) → ndarray

Converts the voltage values of the real cells to the input data for the neural network

**Parameters**

- **ui_new** (*array*) – Voltage values of the cell at each cycle in percentage.
- **Q** (*array*) – Capacity percentages from 0 to 100 from the simulated dataset.
- **size** (*int*) – The length of the curves.
- **material** (*str*) – Chemistry of the cell.

**Returns**

**x_test** – The input data for the neural network.

**Return type**

array

**dataset_files**() → Iterable

Iterable of files included in the dataset.

**feed_pipeline**(*out_dir: str*) → None

Copy selected data file to pipeline step folder.

**Parameters**

**out_dir** – Directory where the dataset was originally stored.

**fetch**(*out_dir: str*) → None

Fetch data to feed the pipeline.

**Parameters**
**out_dir** (*str*) – Directory where the dataset will be stored.

**get_minmaxV**(*material: ~numpy.ndarray) -> (<class 'int'>, <class 'int'>, <class 'str'>*)

Returns the range voltage in which to study the IC curves

**Parameters**
**material** (*numpy array*) – Chemistry to study.

**Returns**
**min_v, max_v, path** – Min and max voltage values and path where data is located,

**Return type**
numpy arrays, str

**is_dataset_ready**() → bool

Check if dataset has been already loaded/downloaded and parsed to package format.

**load**() → None

Parse dataset train/test data to match Conmo's standard.

**normalise_data**(*data: ndarray*, *min_val: float*, *max_val: float*, *low: int = 0*, *high: int = 1*) → float

Normalises the data to the range [low, high]

**Parameters**

- **data** (*numpy array*) – Data to normalise.
- **min** (*float*) – Minimum value of data.
- **max** (*float*) – Maximum value of data.
- **low** (*float*) – Minimum value of the range.
- **high** (*float*) – Maximum value of the range.

**Returns**
**normalised_data** – normalised data

**Return type**
float

**reduce_size**(*ui: ndarray*, *dqi: ndarray*, *size: int*) → ndarray

Reduces the length of the IC data to a given size

**Parameters**

- **ui** (*numpy array*) – Voltage curve.
- **dqi** (*numpy array*) – Derivative of capacity (IC).
- **size** (*int*) – Size at which to reduce the IC data.

**Returns**
Reduced IC.

**Return type**
numpy array

**show_start_message**() → None

Show starting step info message.

**sklearn_predefined_split**() → Iterable[int]

Generates array of indexes of same length as sequences to be used with 'PredefinedSplit'

**Return type**
array, list with the index for each sequence of the dataset.

### Methods

| | |
|---|---|
| *IC*(u, q[, ui_step, minV, maxV]) | Get the ICA data for a given voltage curve |
| *__init__*(path, chemistry, test_set) | Constructor of local dataset. |
| *convert_to_input_data*(ui_new, Q, size, material) | Converts the voltage values of the real cells to the input data for the neural network |
| *dataset_files*() | Iterable of files included in the dataset. |
| *feed_pipeline*(out_dir) | Copy selected data file to pipeline step folder. |
| *fetch*(out_dir) | Fetch data to feed the pipeline. |
| *get_minmaxV*(material) | Returns the range voltage in which to study the IC curves |
| *is_dataset_ready*() | Check if dataset has been already loaded/downloaded and parsed to package format. |
| *load*() | Parse dataset train/test data to match Conmo's standard. |
| *normalise_data*(data, min_val, max_val[, ...]) | Normalises the data to the range [low, high] |
| *reduce_size*(ui, dqi, size) | Reduces the length of the IC data to a given size |
| *show_start_message*() | Show starting step info message. |
| *sklearn_predefined_split*() | Generates array of indexes of same length as sequences to be used with 'PredefinedSplit' |

#### Attributes

| |
|---|
| CHEMISTRY_LIST |
| MAX_V |
| MAX_V_LFP |
| MAX_V_NCA |
| MAX_V_NMC |
| MIN_V |
| MIN_V_LFP |
| MIN_V_NCA |
| MIN_V_NMC |
| SIZE |
| UI_STEP |

### 4.1.3 conmo.splitters

Once the dataset has been loaded, it is necessary to separate the training and test parts. The *conmo.splitters* submodule permits generate new splitters or use predefined ones from the Scikit-Learn library.

| |
|---|
| *splitters.splitter.Splitter*() |
| *splitters.SklearnSplitter*(splitter[, groups]) |

#### conmo.splitters.splitter.Splitter

**class** conmo.splitters.splitter.**Splitter**

**__init__**()

**already_splitted**(*df: DataFrame*) → bool

Checks if the dataset was already splitted.

**Parameters**
**df** (*Pandas Dataframe*) – Input dataset.

**Returns**
True in case the dataset was already splitted, False otherwise.

**Return type**
bool

**Raises**
**RuntimeError** – If the dataset isn't splitted and doesn't follow Conmo's format.

**load_input**(*in_dir: str*) -> (<class 'pandas.core.frame.DataFrame'>, <class 'pandas.core.frame.DataFrame'>)

Read parquet data and labels files of the chosen dataset.

**Parameters**
**in_dir** (*str*) – Input directory where the files are located.

**Returns**

- **data** (*Pandas Dataframe*) – Loaded data file.
- **labels** (*Pandas Dataframe*) – Loaded labels file.

**Raises**
**If data and labels have different sequences values.** –

**save_output**(*out_dir: str*, *data: DataFrame*, *labels: DataFrame*) → None

Save splitted dataset to parquet format.

**Parameters**

- **out_dir** (*str*) – Output directory where the results will be saved.
- **data** (*Pandas Dataframe*) – Splitted data.
- **labels** (*Pandas Dataframe*) – Splitted labels.

**show_start_message**()

Simple method to print on the terminal the name of the selected splitter.

**abstract split**(*in_dir: str*, *out_dir: str*) → None

Performs the split to both data and labels of the dataset.

**Parameters**

- **in_dir** (*str*) – Input directory of the before step.
- **out_dir** (*str*) – Output directory where te split data will be stored.

### Methods

| | |
|---|---|
| *__init__*() | |

| | |
|---|---|
| *already_splitted*(df) | Checks if the dataset was already splitted. |
| *load_input*(in_dir) | Read parquet data and labels files of the chosen dataset. |
| *save_output*(out_dir, data, labels) | Save splitted dataset to parquet format. |
| *show_start_message*() | Simple method to print on the terminal the name of the selected splitter. |
| *split*(in_dir, out_dir) | Performs the split to both data and labels of the dataset. |

### conmo.splitters.SklearnSplitter

class conmo.splitters.**SklearnSplitter**(*splitter: Union[GroupKFold, GroupShuffleSplit, KFold, LeaveOneGroupOut, LeavePGroupsOut, LeaveOneOut, LeavePOut, PredefinedSplit, RepeatedKFold, RepeatedStratifiedKFold, ShuffleSplit, StratifiedKFold, StratifiedShuffleSplit, TimeSeriesSplit]*, *groups: Optional[Iterable[int]] = None*)

**__init__**(*splitter: Union[GroupKFold, GroupShuffleSplit, KFold, LeaveOneGroupOut, LeavePGroupsOut, LeaveOneOut, LeavePOut, PredefinedSplit, RepeatedKFold, RepeatedStratifiedKFold, ShuffleSplit, StratifiedKFold, StratifiedShuffleSplit, TimeSeriesSplit]*, *groups: Optional[Iterable[int]] = None*) → None

**already_splitted**(*df: DataFrame*) → bool

Checks if the dataset was already splitted.

**Parameters**
**df** (*Pandas Dataframe*) – Input dataset.

**Returns**
True in case the dataset was already splitted, False otherwise.

**Return type**
bool

**Raises**
**RuntimeError** – If the dataset isn't splitted and doesn't follow Conmo's format.

**extract_fold**(*df: ~pandas.core.frame.DataFrame*, *sequences: ~numpy.ndarray*, *fold: int*, *train_idx: ~numpy.ndarray*, *test_idx: ~numpy.ndarray*) -> (<class 'numpy.ndarray'>, <class 'numpy.ndarray'>)

**load_input**(*in_dir: str*) -> (<class 'pandas.core.frame.DataFrame'>, <class 'pandas.core.frame.DataFrame'>)

Read parquet data and labels files of the chosen dataset.

**Parameters**
**in_dir** (*str*) – Input directory where the files are located.

**Returns**

- **data** (*Pandas Dataframe*) – Loaded data file.
- **labels** (*Pandas Dataframe*) – Loaded labels file.

**Raises**
**If data and labels have different sequences values.** –

**save_output**(*out_dir: str*, *data: DataFrame*, *labels: DataFrame*) → None

Save splitted dataset to parquet format.

**Parameters**

- **out_dir** (*str*) – Output directory where the results will be saved.
- **data** (*Pandas Dataframe*) – Splitted data.
- **labels** (*Pandas Dataframe*) – Splitted labels.

**show_start_message**()

Simple method to print on the terminal the name of the selected splitter.

**split**(*in_dir: str*, *out_dir: str*) → None

Performs the split to both data and labels of the dataset.

**Parameters**

- **in_dir** (*str*) – Input directory of the before step.
- **out_dir** (*str*) – Output directory where te split data will be stored.

**to_dataframe**(*df: DataFrame*, *data: ndarray*, *index: ndarray*) → DataFrame

### Methods

| | |
|---|---|
| *__init__*(splitter[, groups]) | |
| *already_splitted*(df) | Checks if the dataset was already splitted. |
| *extract_fold*(df, sequences, fold, train_idx, ...) | |
| *load_input*(in_dir) | Read parquet data and labels files of the chosen dataset. |
| *save_output*(out_dir, data, labels) | Save splitted dataset to parquet format. |
| *show_start_message*() | Simple method to print on the terminal the name of the selected splitter. |
| *split*(in_dir, out_dir) | Performs the split to both data and labels of the dataset. |
| *to_dataframe*(df, data, index) | |

## 4.1.4 conmo.preprocesses

The aim of the *conmo.preprocesses* submodule is to apply a series of transformations to the data set before it is used as input to the algorithms. Several types of preprocesses implemented are usually used in time series anomaly detection problems.

| | |
|---|---|
| *preprocesses.preprocess.Preprocess*() | Abstract base class for a Preprocess. |
| *preprocesses.preprocess.ExtendedPreprocess*(...) | Specific class to implement preprocessing which consists of applying certain transformations on some columns of the dataset. |
| *preprocesses.Binarizer*(to_data, to_labels, ...) | |
| *preprocesses.CustomPreprocess*(fn) | Core class used to implement self-created preprocess. |
| *preprocesses.RULImputation*(threshold) | |
| *preprocesses.SavitzkyGolayFilter*(to_data, ...) | |
| *preprocesses.SklearnPreprocess*(to_data, ...) | Class used to wrap existing preprocess in the Scikit-Learn library. |

### conmo.preprocesses.preprocess.Preprocess

**class** conmo.preprocesses.preprocess.**Preprocess**

Abstract base class for a Preprocess.

This class is an abstract class from which other subclasses inherit and must not be instanciated directly.

**__init__**()

**abstract apply**(*in_dir: str*, *out_dir: str*) → None

Applies the preprocess to the given dataset.

**Parameters**

- **in_dir** (*str*) – Input directory where the files are located. Usually, this is the output directory of the splitter step.
- **out_dir** (*str*) – Output directory where the files will be saved.

**load_input**(*in_dir: str*) -> (<*class 'pandas.core.frame.DataFrame'*>, <*class 'pandas.core.frame.DataFrame'*>)

Read parquet data and labels files of the chosen dataset before it's split.

**Parameters**

**in_dir** (*str*) – Input directory where the files are located.

**Returns**

- **data** (*Pandas Dataframe*) – Loaded data file.
- **labels** (*Pandas Dataframe*) – Loaded labels file.

**save_output**(*out_dir: str*, *data: DataFrame*, *labels: DataFrame*) → None

Save preprocessed dataset to parquet format.

**Parameters**

- **out_dir** (*str*) – Output directory where the results will be saved.
- **data** (*Pandas Dataframe*) – Preprocessed data.
- **labels** (*Pandas Dataframe*) – Preprocessed labels.

**show_start_message**() → None

Simple method to print on the terminal the name of the selected splitter.

#### Methods

| | |
|---|---|
| *__init__*() | |
| *apply*(in_dir, out_dir) | Applies the preprocess to the given dataset. |
| *load_input*(in_dir) | Read parquet data and labels files of the chosen dataset before it's split. |
| *save_output*(out_dir, data, labels) | Save preprocessed dataset to parquet format. |
| *show_start_message*() | Simple method to print on the terminal the name of the selected splitter. |

### conmo.preprocesses.preprocess.ExtendedPreprocess

class conmo.preprocesses.preprocess.**ExtendedPreprocess**(*to_data: Union[bool, Iterable[str]]*, *to_labels: Union[bool, Iterable[str]]*, *test_set: bool*)

Specific class to implement preprocessing which consists of applying certain transformations on some columns of the dataset. The preprocessing that inherit from this class have in the constructor to_data, to_labels and test_set to indicate the columns on which to apply the DATA and LABELS preprocessing respectively, and if the TEST ones are included or not.

**__init__**(*to_data: Union[bool, Iterable[str]]*, *to_labels: Union[bool, Iterable[str]]*, *test_set: bool*) → None

**apply**(*in_dir: str*, *out_dir: str*) → None

Applies the preprocess to the given dataset.

**Parameters**

- **in_dir** (*str*) – Input directory where the files are located. Usually, this is the output directory of the splitter step.
- **out_dir** (*str*) – Output directory where the files will be saved.

**extract_columns**(*df: DataFrame*, *columns: Union[bool, Iterable[str]]*) → Iterable[str]

Returns a list containig all the column's name of the data.

**Parameters**

- **df** (*Pandas Dataframe*) – Dataframe containing the data.
- **columns** (*Union[bool, Iterable[str]]*) – Bool value if the dataframe has columns or the list of columns.

**Returns**

**columns** – List containing the names of the dataframe's columns.

**Return type**

Iterable[str]

**load_input**(*in_dir: str) -> (<class 'pandas.core.frame.DataFrame'>*, *<class 'pandas.core.frame.DataFrame'>*)

Read parquet data and labels files of the chosen dataset before it's split.

**Parameters**

**in_dir** (*str*) – Input directory where the files are located.

**Returns**

- **data** (*Pandas Dataframe*) – Loaded data file.
- **labels** (*Pandas Dataframe*) – Loaded labels file.

**save_output**(*out_dir: str*, *data: DataFrame*, *labels: DataFrame*) → None

Save preprocessed dataset to parquet format.

**Parameters**

- **out_dir** (*str*) – Output directory where the results will be saved.
- **data** (*Pandas Dataframe*) – Preprocessed data.
- **labels** (*Pandas Dataframe*) – Preprocessed labels.

**show_start_message**() → None

Simple method to print on the terminal the name of the selected splitter.

**abstract transform**(*df: DataFrame*, *columns: Iterable[str]*) → DataFrame

Performs the preprocess over the dataframe with the given columns.

**Parameters**

- **df** (*Pandas Dataframe*) – Dataframe containing the data or the labels of the dataset.
- **columns** (*Iterable[str]*) – List of columns that will be used in the preprocess. Also the columns of the final dataframe.

**Returns**

Dataframe preprocessed.

**Return type**

Pandas Dataframe

### Methods

| | |
|---|---|
| *__init__*(to_data, to_labels, test_set) | |

| | |
|---|---|
| *apply*(in_dir, out_dir) | Applies the preprocess to the given dataset. |
| *extract_columns*(df, columns) | Returns a list containig all the column's name of the data. |
| *load_input*(in_dir) | Read parquet data and labels files of the chosen dataset before it's split. |
| *save_output*(out_dir, data, labels) | Save preprocessed dataset to parquet format. |
| *show_start_message*() | Simple method to print on the terminal the name of the selected splitter. |
| *transform*(df, columns) | Performs the preprocess over the dataframe with the given columns. |

## conmo.preprocesses.Binarizer

**class** conmo.preprocesses.**Binarizer**(*to_data: Union[bool, Iterable[str]]*, *to_labels: Union[bool, Iterable[str]]*, *test_set: bool*, *threshold: int*)

**__init__**(*to_data: Union[bool, Iterable[str]]*, *to_labels: Union[bool, Iterable[str]]*, *test_set: bool*, *threshold: int*) → None

**apply**(*in_dir: str*, *out_dir: str*) → None

Applies the preprocess to the given dataset.

**Parameters**

- **in_dir** (*str*) – Input directory where the files are located. Usually, this is the output directory of the splitter step.
- **out_dir** (*str*) – Output directory where the files will be saved.

**extract_columns**(*df: DataFrame*, *columns: Union[bool, Iterable[str]]*) → Iterable[str]

Returns a list containig all the column's name of the data.

**Parameters**

- **df** (*Pandas Dataframe*) – Dataframe containing the data.
- **columns** (*Union[bool, Iterable[str]]*) – Bool value if the dataframe has columns or the list of columns.

**Returns**
**columns** – List containing the names of the dataframe's columns.

**Return type**
Iterable[str]

**load_input**(*in_dir: str*) -> (<class 'pandas.core.frame.DataFrame'>, <class 'pandas.core.frame.DataFrame'>)

Read parquet data and labels files of the chosen dataset before it's split.

**Parameters**
**in_dir** (*str*) – Input directory where the files are located.

**Returns**

- **data** (*Pandas Dataframe*) – Loaded data file.
- **labels** (*Pandas Dataframe*) – Loaded labels file.

**save_output**(*out_dir: str*, *data: DataFrame*, *labels: DataFrame*) → None

Save preprocessed dataset to parquet format.

**Parameters**

- **out_dir** (*str*) – Output directory where the results will be saved.
- **data** (*Pandas Dataframe*) – Preprocessed data.
- **labels** (*Pandas Dataframe*) – Preprocessed labels.

**show_start_message**() → None

Simple method to print on the terminal the name of the selected splitter.

**transform**(*df: DataFrame*, *columns: Iterable[str]*) → DataFrame

Performs the preprocess over the dataframe with the given columns.

**Parameters**

- **df** (*Pandas Dataframe*) – Dataframe containing the data or the labels of the dataset.
- **columns** (*Iterable[str]*) – List of columns that will be used in the preprocess. Also the columns of the final dataframe.

**Returns**
Dataframe preprocessed.

**Return type**
Pandas Dataframe

### Methods

| | |
|---|---|
| *__init__*(to_data, to_labels, test_set, threshold) | |

| | |
|---|---|
| *apply*(in_dir, out_dir) | Applies the preprocess to the given dataset. |
| *extract_columns*(df, columns) | Returns a list containig all the column's name of the data. |
| *load_input*(in_dir) | Read parquet data and labels files of the chosen dataset before it's split. |
| *save_output*(out_dir, data, labels) | Save preprocessed dataset to parquet format. |
| *show_start_message*() | Simple method to print on the terminal the name of the selected splitter. |
| *transform*(df, columns) | Performs the preprocess over the dataframe with the given columns. |

## conmo.preprocesses.CustomPreprocess

**class** conmo.preprocesses.**CustomPreprocess**(*fn: Callable[[DataFrame, DataFrame], Tuple[DataFrame, DataFrame]]*)

Core class used to implement self-created preprocess. Such preprocess will be wrapped in a function that will be passed as an argument to the constructor of this class.

**__init__**(*fn: Callable[[DataFrame, DataFrame], Tuple[DataFrame, DataFrame]]*) → None

**apply**(*in_dir: str*, *out_dir: str*) → None

Applies the custom preprocess to labels and data.

**Parameters**

- **in_dir** (*str*) – Input directory where the files are located. Usually, this is the output directory of the splitter step.
- **out_dir** (*str*) – Output directory where the files will be saved.

**load_input**(*in_dir: str) -> (<class 'pandas.core.frame.DataFrame'>*, *<class 'pandas.core.frame.DataFrame'>*)

Read parquet data and labels files of the chosen dataset before it's split.

**Parameters**

**in_dir** (*str*) – Input directory where the files are located.

**Returns**

- **data** (*Pandas Dataframe*) – Loaded data file.
- **labels** (*Pandas Dataframe*) – Loaded labels file.

**save_output**(*out_dir: str*, *data: DataFrame*, *labels: DataFrame*) → None

Save preprocessed dataset to parquet format.

**Parameters**

- **out_dir** (*str*) – Output directory where the results will be saved.
- **data** (*Pandas Dataframe*) – Preprocessed data.
- **labels** (*Pandas Dataframe*) – Preprocessed labels.

**show_start_message**() → None

Simple method to print on the terminal the name of the selected splitter.

### Methods

| | |
|---|---|
| *__init__*(fn) | |

| | |
|---|---|
| *apply*(in_dir, out_dir) | Applies the custom preprocess to labels and data. |
| *load_input*(in_dir) | Read parquet data and labels files of the chosen dataset before it's split. |
| *save_output*(out_dir, data, labels) | Save preprocessed dataset to parquet format. |
| *show_start_message*() | Simple method to print on the terminal the name of the selected splitter. |

## conmo.preprocesses.RULImputation

**class** conmo.preprocesses.**RULImputation**(*threshold: int*)

**__init__**(*threshold: int*) → None

**apply**(*in_dir*, *out_dir*) → None

RUL imputation per TIME sample based on SEQUENCE labels, generating labels for each TIME.

**load_input**(*in_dir: str*) -> (<class 'pandas.core.frame.DataFrame'>, <class 'pandas.core.frame.DataFrame'>)

Read parquet data and labels files of the chosen dataset before it's split.

**Parameters**

**in_dir** (*str*) – Input directory where the files are located.

**Returns**

- **data** (*Pandas Dataframe*) – Loaded data file.
- **labels** (*Pandas Dataframe*) – Loaded labels file.

**save_output**(*out_dir: str*, *data: DataFrame*, *labels: DataFrame*) → None

Save preprocessed dataset to parquet format.

**Parameters**

- **out_dir** (*str*) – Output directory where the results will be saved.
- **data** (*Pandas Dataframe*) – Preprocessed data.
- **labels** (*Pandas Dataframe*) – Preprocessed labels.

**show_start_message**() → None

Simple method to print on the terminal the name of the selected splitter.

### Methods

| | |
|---|---|
| *__init__*(threshold) | |

| | |
|---|---|
| *apply*(in_dir, out_dir) | RUL imputation per TIME sample based on SEQUENCE labels, generating labels for each TIME. |
| *load_input*(in_dir) | Read parquet data and labels files of the chosen dataset before it's split. |
| *save_output*(out_dir, data, labels) | Save preprocessed dataset to parquet format. |
| *show_start_message*() | Simple method to print on the terminal the name of the selected splitter. |

## conmo.preprocesses.SavitzkyGolayFilter

**class** conmo.preprocesses.**SavitzkyGolayFilter**(*to_data: Union[bool, Iterable[str]]*, *to_labels: Union[bool, Iterable[str]]*, *test_set: bool*, *window_length: int*, *polyorder: int*, *deriv: Optional[int] = 0*, *delta: Optional[float] = 1.0*, *mode: Optional[str] = 'interp'*, *cval: Optional[float] = 0.0*)

**__init__**(*to_data: Union[bool, Iterable[str]]*, *to_labels: Union[bool, Iterable[str]]*, *test_set: bool*, *window_length: int*, *polyorder: int*, *deriv: Optional[int] = 0*, *delta: Optional[float] = 1.0*, *mode: Optional[str] = 'interp'*, *cval: Optional[float] = 0.0*)

**apply**(*in_dir: str*, *out_dir: str*) → None

Applies the preprocess to the given dataset.

**Parameters**

- **in_dir** (*str*) – Input directory where the files are located. Usually, this is the output directory of the splitter step.
- **out_dir** (*str*) – Output directory where the files will be saved.

**extract_columns**(*df: DataFrame*, *columns: Union[bool, Iterable[str]]*) → Iterable[str]

Returns a list containig all the column's name of the data.

**Parameters**

- **df** (*Pandas Dataframe*) – Dataframe containing the data.
- **columns** (*Union[bool, Iterable[str]]*) – Bool value if the dataframe has columns or the list of columns.

**Returns**

**columns** – List containing the names of the dataframe's columns.

**Return type**

Iterable[str]

**load_input**(*in_dir: str) -> (<class 'pandas.core.frame.DataFrame'>*, *<class 'pandas.core.frame.DataFrame'>*)

Read parquet data and labels files of the chosen dataset before it's split.

**Parameters**

**in_dir** (*str*) – Input directory where the files are located.

**Returns**

- **data** (*Pandas Dataframe*) – Loaded data file.
- **labels** (*Pandas Dataframe*) – Loaded labels file.

**save_output**(*out_dir: str*, *data: DataFrame*, *labels: DataFrame*) → None

Save preprocessed dataset to parquet format.

**Parameters**

- **out_dir** (*str*) – Output directory where the results will be saved.
- **data** (*Pandas Dataframe*) – Preprocessed data.
- **labels** (*Pandas Dataframe*) – Preprocessed labels.

**show_start_message**() → None

Simple method to print on the terminal the name of the selected splitter.

**transform**(*df: DataFrame*, *columns: Iterable[str]*) → DataFrame

Performs the preprocess over the dataframe with the given columns.

**Parameters**

- **df** (*Pandas Dataframe*) – Dataframe containing the data or the labels of the dataset.
- **columns** (*Iterable[str]*) – List of columns that will be used in the preprocess. Also the columns of the final dataframe.

**Returns**

Dataframe preprocessed.

**Return type**

Pandas Dataframe

### Methods

| | |
|---|---|
| *__init__*(to_data, to_labels, test_set, ...) | |
| *apply*(in_dir, out_dir) | Applies the preprocess to the given dataset. |
| *extract_columns*(df, columns) | Returns a list containig all the column's name of the data. |
| *load_input*(in_dir) | Read parquet data and labels files of the chosen dataset before it's split. |
| *save_output*(out_dir, data, labels) | Save preprocessed dataset to parquet format. |
| *show_start_message*() | Simple method to print on the terminal the name of the selected splitter. |
| *transform*(df, columns) | Performs the preprocess over the dataframe with the given columns. |

### conmo.preprocesses.SklearnPreprocess

class conmo.preprocesses.**SklearnPreprocess**(*to_data: Union[bool, Iterable[str]]*, *to_labels: Union[bool, Iterable[str]]*, *test_set: bool*, *preprocess: Union[Binarizer, FunctionTransformer, KBinsDiscretizer, KernelCenterer, LabelBinarizer, LabelEncoder, MultiLabelBinarizer, MaxAbsScaler, MinMaxScaler, Normalizer, OneHotEncoder, OrdinalEncoder, PolynomialFeatures, PowerTransformer, QuantileTransformer, RobustScaler, StandardScaler]*)

Class used to wrap existing preprocess in the Scikit-Learn library. It also allows this preprocess to be applied to certain columns of the dataset.

**__init__**(*to_data: Union[bool, Iterable[str]]*, *to_labels: Union[bool, Iterable[str]]*, *test_set: bool*, *preprocess: Union[Binarizer, FunctionTransformer, KBinsDiscretizer, KernelCenterer, LabelBinarizer, LabelEncoder, MultiLabelBinarizer, MaxAbsScaler, MinMaxScaler, Normalizer, OneHotEncoder, OrdinalEncoder, PolynomialFeatures, PowerTransformer, QuantileTransformer, RobustScaler, StandardScaler]*) → None

**apply**(*in_dir: str*, *out_dir: str*) → None

Applies the preprocess to the given dataset.

**Parameters**

- **in_dir** (*str*) – Input directory where the files are located. Usually, this is the output directory of the splitter step.
- **out_dir** (*str*) – Output directory where the files will be saved.

**extract_columns**(*df: DataFrame*, *columns: Union[bool, Iterable[str]]*) → Iterable[str]

Returns a list containig all the column's name of the data.

**Parameters**

- **df** (*Pandas Dataframe*) – Dataframe containing the data.
- **columns** (*Union[bool, Iterable[str]]*) – Bool value if the dataframe has columns or the list of columns.

**Returns**

**columns** – List containing the names of the dataframe's columns.

**Return type**

Iterable[str]

**load_input**(*in_dir: str) -> (<class 'pandas.core.frame.DataFrame'>*, *<class 'pandas.core.frame.DataFrame'>*)

Read parquet data and labels files of the chosen dataset before it's split.

**Parameters**

**in_dir** (*str*) – Input directory where the files are located.

**Returns**

- **data** (*Pandas Dataframe*) – Loaded data file.
- **labels** (*Pandas Dataframe*) – Loaded labels file.

**save_output**(*out_dir: str*, *data: DataFrame*, *labels: DataFrame*) → None

Save preprocessed dataset to parquet format.

**Parameters**

- **out_dir** (*str*) – Output directory where the results will be saved.
- **data** (*Pandas Dataframe*) – Preprocessed data.
- **labels** (*Pandas Dataframe*) – Preprocessed labels.

**show_start_message**() → None

Simple method to print on the terminal the name of the selected splitter.

**transform**(*df: DataFrame*, *columns: Iterable[str]*) → DataFrame

Performs the preprocess over the dataframe with the given columns.

**Parameters**

- **df** (*Pandas Dataframe*) – Dataframe containing the data or the labels of the dataset.
- **columns** (*Iterable[str]*) – List of columns that will be used in the preprocess. Also the columns of the final dataframe.

**Returns**

Dataframe preprocessed.

**Return type**

Pandas Dataframe

#### Methods

| | |
|---|---|
| *__init__*(to_data, to_labels, test_set, ...) | |
| *apply*(in_dir, out_dir) | Applies the preprocess to the given dataset. |
| *extract_columns*(df, columns) | Returns a list containig all the column's name of the data. |
| *load_input*(in_dir) | Read parquet data and labels files of the chosen dataset before it's split. |
| *save_output*(out_dir, data, labels) | Save preprocessed dataset to parquet format. |
| *show_start_message*() | Simple method to print on the terminal the name of the selected splitter. |
| *transform*(df, columns) | Performs the preprocess over the dataframe with the given columns. |

### 4.1.5 conmo.algorithms

The *conmo.algorithms* submodule contains everything related to algorithms in Conmo, from abstract classes to introduce new algorithms in Conmo to implementations of some of the algorithms used in the example experiments.

| | |
|---|---|
| *algorithms.algorithm.Algorithm*() | |
| *algorithms.algorithm.AnomalyDetectionThresholdBasedAlgorithm*(...) | |
| *algorithms.algorithm.AnomalyDetectionClassBasedAlgorithm*() | |
| *algorithms.PCAMahalanobis*([n_components, ...]) | |
| *algorithms.OneClassSVM*([kernel, degree, ...]) | |
| *algorithms.KerasAutoencoder*([encoding_dim, ...]) | |
| *algorithms.PretrainedRandomForest*(pretrained) | |
| *algorithms.PretrainedMultilayerPerceptron*(...) | |
| *algorithms.PretrainedCNN1D*(pretrained, input_len) | |

### conmo.algorithms.algorithm.Algorithm

**class** conmo.algorithms.algorithm.**Algorithm**

**__init__**()

**execute**(*idx: int*, *in_dir: str*, *out_dir: str*) → str

Performs a complete execution of the algorithm, loading input data, performing a run through the folds and saving the results.

**Parameters**

- **idx** (*int*) – Index of the algorithm in the Experiment. Userful in case you want to experiment with several algorithms.
- **in_dir** (*str*) – Intermediate directory where the input data to the algorithm is stored.
- **out_dir** (*str*) – Intermediate directory where the output data (predictios of the algorithm) will be stored.

**Returns**
Name of the output directory.

**Return type**
str

**abstract fit_predict**(*data_train: DataFrame*, *data_test: DataFrame*, *labels_train: DataFrame*, *labels_test: DataFrame*) → DataFrame

Trains the model with train data and then performs predictions with the trained algorithm over the test data.

**Parameters**

- **data_train** (*Pandas Dataframe*) – Train data.
- **data_test** (*Pandas Dataframe*) – Test data.
- **labels_train** (*Pandas Dataframe*) – Train labels.

- **labels_test** (*Pandas Dataframe*) – Test labels.

**Returns**
Results of the predictions made on the test set.

**Return type**
Pandas Dataframe

**labels_per_sequence**(*labels: DataFrame*) → bool

Use only with time series datasets. Checks if the labels file of the chosen dataset has an index format with sequences only or sequences and time. *This method in future updates will be changed to a specific class for time series.*

**Parameters**
**labels** (*Pandas Dataframe*) – Labels file of the dataset.

**Returns**
True if the labels contains 1 level of index with sequence or False if the labels file contains 2 leves with sequence and time.

**Return type**
bool

**Raises**
**RuntimeError** – If the number of index levels is invalid.

**load_input**(*in_dir: str) -> (<class 'pandas.core.frame.DataFrame'>, <class 'pandas.core.frame.DataFrame'>*)

Read parquet data and labels files of the chosen dataset.

**Parameters**
**in_dir** (*str*) – Input directory where the files are located.

**Returns**

- **data** (*Pandas Dataframe*) – Loaded data file.
- **labels** (*Pandas Dataframe*) – Loaded labels file.

**save_output**(*results: DataFrame*, *out_dir: str*, *idx: int*) → str

Save algorithms output to parquet format.

**Parameters**

- **results** (*Pandas Dataframe*) – Dataframe with the results of the execution.
- **out_dir** (*str*) – Output directory where the results will be saved.
- **idx** (*int*) – Index of the algorithm in the Experiment. Userful in case you want to experiment with several algorithms.

**show_start_message**()

Simple method to print on the terminal the name of the algorithm to be executed.

### Methods

| | |
|---|---|
| *__init__*() | |

| | |
|---|---|
| *execute*(idx, in_dir, out_dir) | Performs a complete execution of the algorithm, loading input data, performing a run through the folds and saving the results. |
| *fit_predict*(data_train, data_test, ...) | Trains the model with train data and then performs predictions with the trained algorithm over the test data. |
| *labels_per_sequence*(labels) | Use only with time series datasets. |
| *load_input*(in_dir) | Read parquet data and labels files of the chosen dataset. |
| *save_output*(results, out_dir, idx) | Save algorithms output to parquet format. |
| *show_start_message*() | Simple method to print on the terminal the name of the algorithm to be executed. |

## conmo.algorithms.algorithm.AnomalyDetectionThresholdBasedAlgorithm

**class** conmo.algorithms.algorithm.**AnomalyDetectionThresholdBasedAlgorithm**(*threshold_mode: str*, *threshold_value: float*)

**__init__**(*threshold_mode: str*, *threshold_value: float*)

**execute**(*idx: int*, *in_dir: str*, *out_dir: str*) → str

Performs a complete execution of the algorithm, loading input data, performing a run through the folds and saving the results.

**Parameters**

- **idx** (*int*) – Index of the algorithm in the Experiment. Userful in case you want to experiment with several algorithms.
- **in_dir** (*str*) – Intermediate directory where the input data to the algorithm is stored.
- **out_dir** (*str*) – Intermediate directory where the output data (predictios of the algorithm) will be stored.

**Returns**
Name of the output directory.

**Return type**
str

**find_anomaly_threshold**(*values: ndarray*) → float

Finds anomaly threshold for threshold based algorithms. 3 different approaches are currently implemented.

**Parameters**
**values** (*Numpy ndarray*) – Results of the algoritm execution.

**Returns**
Calculated threshold value.

**Return type**
float

**abstract fit_predict**(*data_train: DataFrame*, *data_test: DataFrame*, *labels_train: DataFrame*, *labels_test: DataFrame*) → DataFrame

Trains the model with train data and then performs predictions with the trained algorithm over the test data.

**Parameters**

- **data_train** (*Pandas Dataframe*) – Train data.
- **data_test** (*Pandas Dataframe*) – Test data.
- **labels_train** (*Pandas Dataframe*) – Train labels.
- **labels_test** (*Pandas Dataframe*) – Test labels.

**Returns**
Results of the predictions made on the test set.

**Return type**
Pandas Dataframe

**labels_per_sequence**(*labels: DataFrame*) → bool

Use only with time series datasets. Checks if the labels file of the chosen dataset has an index format with sequences only or sequences and time. *This method in future updates will be changed to a specific class for time series.*

**Parameters**
**labels** (*Pandas Dataframe*) – Labels file of the dataset.

**Returns**
True if the labels contains 1 level of index with sequence or False if the labels file contains 2 leves with sequence and time.

**Return type**
bool

**Raises**
**RuntimeError** – If the number of index levels is invalid.

**load_input**(*in_dir: str) -> (<class 'pandas.core.frame.DataFrame'>*, *<class 'pandas.core.frame.DataFrame'>*)

Read parquet data and labels files of the chosen dataset.

**Parameters**
**in_dir** (*str*) – Input directory where the files are located.

**Returns**

- **data** (*Pandas Dataframe*) – Loaded data file.
- **labels** (*Pandas Dataframe*) – Loaded labels file.

**save_output**(*results: DataFrame*, *out_dir: str*, *idx: int*) → str

Save algorithms output to parquet format.

**Parameters**

- **results** (*Pandas Dataframe*) – Dataframe with the results of the execution.
- **out_dir** (*str*) – Output directory where the results will be saved.
- **idx** (*int*) – Index of the algorithm in the Experiment. Userful in case you want to experiment with several algorithms.

**show_start_message**()

Simple method to print on the terminal the name of the algorithm to be executed.

### Methods

| | |
|---|---|
| *__init__*(threshold_mode, threshold_value) | |
| *execute*(idx, in_dir, out_dir) | Performs a complete execution of the algorithm, loading input data, performing a run through the folds and saving the results. |
| *find_anomaly_threshold*(values) | Finds anomaly threshold for threshold based algorithms. |
| *fit_predict*(data_train, data_test, ...) | Trains the model with train data and then performs predictions with the trained algorithm over the test data. |
| *labels_per_sequence*(labels) | Use only with time series datasets. |
| *load_input*(in_dir) | Read parquet data and labels files of the chosen dataset. |
| *save_output*(results, out_dir, idx) | Save algorithms output to parquet format. |
| *show_start_message*() | Simple method to print on the terminal the name of the algorithm to be executed. |

## conmo.algorithms.algorithm.AnomalyDetectionClassBasedAlgorithm

**class** conmo.algorithms.algorithm.**AnomalyDetectionClassBasedAlgorithm**

**__init__**()

**execute**(*idx: int*, *in_dir: str*, *out_dir: str*) → str

Performs a complete execution of the algorithm, loading input data, performing a run through the folds and saving the results.

**Parameters**

- **idx** (*int*) – Index of the algorithm in the Experiment. Userful in case you want to experiment with several algorithms.
- **in_dir** (*str*) – Intermediate directory where the input data to the algorithm is stored.
- **out_dir** (*str*) – Intermediate directory where the output data (predictios of the algorithm) will be stored.

**Returns**

Name of the output directory.

**Return type**

str

**abstract fit_predict**(*data_train: DataFrame*, *data_test: DataFrame*, *labels_train: DataFrame*, *labels_test: DataFrame*) → DataFrame

Trains the model with train data and then performs predictions with the trained algorithm over the test data.

**Parameters**

- **data_train** (*Pandas Dataframe*) – Train data.
- **data_test** (*Pandas Dataframe*) – Test data.
- **labels_train** (*Pandas Dataframe*) – Train labels.
- **labels_test** (*Pandas Dataframe*) – Test labels.

**Returns**
Results of the predictions made on the test set.

**Return type**
Pandas Dataframe

**labels_per_sequence**(*labels: DataFrame*) → bool

Use only with time series datasets. Checks if the labels file of the chosen dataset has an index format with sequences only or sequences and time. *This method in future updates will be changed to a specific class for time series.*

**Parameters**
**labels** (*Pandas Dataframe*) – Labels file of the dataset.

**Returns**
True if the labels contains 1 level of index with sequence or False if the labels file contains 2 leves with sequence and time.

**Return type**
bool

**Raises**
**RuntimeError** – If the number of index levels is invalid.

**load_input**(*in_dir: str) -> (<class 'pandas.core.frame.DataFrame'>, <class 'pandas.core.frame.DataFrame'>*)

Read parquet data and labels files of the chosen dataset.

**Parameters**
**in_dir** (*str*) – Input directory where the files are located.

**Returns**

- **data** (*Pandas Dataframe*) – Loaded data file.
- **labels** (*Pandas Dataframe*) – Loaded labels file.

**save_output**(*results: DataFrame*, *out_dir: str*, *idx: int*) → str

Save algorithms output to parquet format.

**Parameters**

- **results** (*Pandas Dataframe*) – Dataframe with the results of the execution.
- **out_dir** (*str*) – Output directory where the results will be saved.
- **idx** (*int*) – Index of the algorithm in the Experiment. Userful in case you want to experiment with several algorithms.

**show_start_message**()

Simple method to print on the terminal the name of the algorithm to be executed.

### Methods

| | |
|---|---|
| *__init__*() | |

| | |
|---|---|
| *execute*(idx, in_dir, out_dir) | Performs a complete execution of the algorithm, loading input data, performing a run through the folds and saving the results. |
| *fit_predict*(data_train, data_test, ...) | Trains the model with train data and then performs predictions with the trained algorithm over the test data. |
| *labels_per_sequence*(labels) | Use only with time series datasets. |
| *load_input*(in_dir) | Read parquet data and labels files of the chosen dataset. |
| *save_output*(results, out_dir, idx) | Save algorithms output to parquet format. |
| *show_start_message*() | Simple method to print on the terminal the name of the algorithm to be executed. |

## conmo.algorithms.PCAMahalanobis

**class** conmo.algorithms.**PCAMahalanobis**(*n_components: float = 0.95*, *robust_estimator: bool = False*, *threshold_mode: str = 'chi2'*, *threshold_value: Optional[Union[int, float]] = 0.95*)

**__init__**(*n_components: float = 0.95*, *robust_estimator: bool = False*, *threshold_mode: str = 'chi2'*, *threshold_value: Optional[Union[int, float]] = 0.95*)

**execute**(*idx: int*, *in_dir: str*, *out_dir: str*) → str

Performs a complete execution of the algorithm, loading input data, performing a run through the folds and saving the results.

**Parameters**

- **idx** (*int*) – Index of the algorithm in the Experiment. Userful in case you want to experiment with several algorithms.
- **in_dir** (*str*) – Intermediate directory where the input data to the algorithm is stored.
- **out_dir** (*str*) – Intermediate directory where the output data (predictios of the algorithm) will be stored.

**Returns**
Name of the output directory.

**Return type**
str

**find_anomaly_threshold**(*values: ndarray*, *n_features: int*) → float

Finds anomaly threshold for threshold based algorithms. 3 different approaches are currently implemented.

**Parameters**
**values** (*Numpy ndarray*) – Results of the algoritm execution.

**Returns**
Calculated threshold value.

**Return type**
float

**fit_predict**(*data_train: DataFrame*, *data_test: DataFrame*, *labels_train: DataFrame*, *labels_test: DataFrame*) → DataFrame

Trains the model with train data and then performs predictions with the trained algorithm over the test data.

**Parameters**

- **data_train** (*Pandas Dataframe*) – Train data.
- **data_test** (*Pandas Dataframe*) – Test data.
- **labels_train** (*Pandas Dataframe*) – Train labels.
- **labels_test** (*Pandas Dataframe*) – Test labels.

**Returns**
Results of the predictions made on the test set.

**Return type**
Pandas Dataframe

**labels_per_sequence**(*labels: DataFrame*) → bool

Use only with time series datasets. Checks if the labels file of the chosen dataset has an index format with sequences only or sequences and time. *This method in future updates will be changed to a specific class for time series.*

**Parameters**
**labels** (*Pandas Dataframe*) – Labels file of the dataset.

**Returns**
True if the labels contains 1 level of index with sequence or False if the labels file contains 2 leves with sequence and time.

**Return type**
bool

**Raises**
**RuntimeError** – If the number of index levels is invalid.

**load_input**(*in_dir: str) -> (<class 'pandas.core.frame.DataFrame'>*, *<class 'pandas.core.frame.DataFrame'>*)

Read parquet data and labels files of the chosen dataset.

**Parameters**
**in_dir** (*str*) – Input directory where the files are located.

**Returns**

- **data** (*Pandas Dataframe*) – Loaded data file.
- **labels** (*Pandas Dataframe*) – Loaded labels file.

**save_output**(*results: DataFrame*, *out_dir: str*, *idx: int*) → str

Save algorithms output to parquet format.

**Parameters**

- **results** (*Pandas Dataframe*) – Dataframe with the results of the execution.
- **out_dir** (*str*) – Output directory where the results will be saved.

- **idx** (*int*) – Index of the algorithm in the Experiment. Userful in case you want to experiment with several algorithms.

**show_start_message**()

Simple method to print on the terminal the name of the algorithm to be executed.

### Methods

| | |
|---|---|
| *__init__*([n_components, robust_estimator, ...]) | |
| *execute*(idx, in_dir, out_dir) | Performs a complete execution of the algorithm, loading input data, performing a run through the folds and saving the results. |
| *find_anomaly_threshold*(values, n_features) | Finds anomaly threshold for threshold based algorithms. |
| *fit_predict*(data_train, data_test, ...) | Trains the model with train data and then performs predictions with the trained algorithm over the test data. |
| *labels_per_sequence*(labels) | Use only with time series datasets. |
| *load_input*(in_dir) | Read parquet data and labels files of the chosen dataset. |
| *save_output*(results, out_dir, idx) | Save algorithms output to parquet format. |
| *show_start_message*() | Simple method to print on the terminal the name of the algorithm to be executed. |

## conmo.algorithms.OneClassSVM

**class** conmo.algorithms.**OneClassSVM**(*kernel: str = 'rbf'*, *degree: int = 3*, *gamma: Union[str, float] = 'scale'*, *coef0: float = 0.0*, *tol: float = 0.001*, *nu: float = 0.5*)

**__init__**(*kernel: str = 'rbf'*, *degree: int = 3*, *gamma: Union[str, float] = 'scale'*, *coef0: float = 0.0*, *tol: float = 0.001*, *nu: float = 0.5*)

**execute**(*idx: int*, *in_dir: str*, *out_dir: str*) → str

Performs a complete execution of the algorithm, loading input data, performing a run through the folds and saving the results.

**Parameters**

- **idx** (*int*) – Index of the algorithm in the Experiment. Userful in case you want to experiment with several algorithms.
- **in_dir** (*str*) – Intermediate directory where the input data to the algorithm is stored.
- **out_dir** (*str*) – Intermediate directory where the output data (predictios of the algorithm) will be stored.

**Returns**

Name of the output directory.

**Return type**

str

**fit_predict**(*data_train: DataFrame*, *data_test: DataFrame*, *labels_train: DataFrame*, *labels_test: DataFrame*) → DataFrame

Trains the model with train data and then performs predictions with the trained algorithm over the test data.

**Parameters**

- **data_train** (*Pandas Dataframe*) – Train data.
- **data_test** (*Pandas Dataframe*) – Test data.
- **labels_train** (*Pandas Dataframe*) – Train labels.
- **labels_test** (*Pandas Dataframe*) – Test labels.

**Returns**
Results of the predictions made on the test set.

**Return type**
Pandas Dataframe

**labels_per_sequence**(*labels: DataFrame*) → bool

Use only with time series datasets. Checks if the labels file of the chosen dataset has an index format with sequences only or sequences and time. *This method in future updates will be changed to a specific class for time series.*

**Parameters**
**labels** (*Pandas Dataframe*) – Labels file of the dataset.

**Returns**
True if the labels contains 1 level of index with sequence or False if the labels file contains 2 leves with sequence and time.

**Return type**
bool

**Raises**
**RuntimeError** – If the number of index levels is invalid.

**load_input**(*in_dir: str) -> (<class 'pandas.core.frame.DataFrame'>*, *<class 'pandas.core.frame.DataFrame'>*)

Read parquet data and labels files of the chosen dataset.

**Parameters**
**in_dir** (*str*) – Input directory where the files are located.

**Returns**

- **data** (*Pandas Dataframe*) – Loaded data file.
- **labels** (*Pandas Dataframe*) – Loaded labels file.

**save_output**(*results: DataFrame*, *out_dir: str*, *idx: int*) → str

Save algorithms output to parquet format.

**Parameters**

- **results** (*Pandas Dataframe*) – Dataframe with the results of the execution.
- **out_dir** (*str*) – Output directory where the results will be saved.
- **idx** (*int*) – Index of the algorithm in the Experiment. Userful in case you want to experiment with several algorithms.

**show_start_message**()

Simple method to print on the terminal the name of the algorithm to be executed.

### Methods

| | |
|---|---|
| *__init__*([kernel, degree, gamma, coef0, tol, nu]) | |

| | |
|---|---|
| *execute*(idx, in_dir, out_dir) | Performs a complete execution of the algorithm, loading input data, performing a run through the folds and saving the results. |
| *fit_predict*(data_train, data_test, ...) | Trains the model with train data and then performs predictions with the trained algorithm over the test data. |
| *labels_per_sequence*(labels) | Use only with time series datasets. |
| *load_input*(in_dir) | Read parquet data and labels files of the chosen dataset. |
| *save_output*(results, out_dir, idx) | Save algorithms output to parquet format. |
| *show_start_message*() | Simple method to print on the terminal the name of the algorithm to be executed. |

## conmo.algorithms.KerasAutoencoder

**class** conmo.algorithms.**KerasAutoencoder**(*encoding_dim: int = 32*, *optimizer: str = 'Adam'*, *loss_f: str = 'mse'*, *epochs: int = 2*, *batch_size: int = 64*, *random_seed: int = 11*, *threshold_mode: str = 'chi2'*, *threshold_value: Optional[Union[int, float]] = 0.95*)

**__init__**(*encoding_dim: int = 32*, *optimizer: str = 'Adam'*, *loss_f: str = 'mse'*, *epochs: int = 2*, *batch_size: int = 64*, *random_seed: int = 11*, *threshold_mode: str = 'chi2'*, *threshold_value: Optional[Union[int, float]] = 0.95*)

**execute**(*idx: int*, *in_dir: str*, *out_dir: str*) → str

Performs a complete execution of the algorithm, loading input data, performing a run through the folds and saving the results.

**Parameters**

- **idx** (*int*) – Index of the algorithm in the Experiment. Userful in case you want to experiment with several algorithms.
- **in_dir** (*str*) – Intermediate directory where the input data to the algorithm is stored.
- **out_dir** (*str*) – Intermediate directory where the output data (predictios of the algorithm) will be stored.

**Returns**

Name of the output directory.

**Return type**

str

**find_anomaly_threshold**(*values: ndarray*, *n_features: int*) → float

Finds anomaly threshold for threshold based algorithms. 3 different approaches are currently implemented.

**Parameters**
**values** (*Numpy ndarray*) – Results of the algoritm execution.

**Returns**
Calculated threshold value.

**Return type**
float

**fit_predict**(*data_train: DataFrame*, *data_test: DataFrame*, *labels_train: DataFrame*, *labels_test: DataFrame*) → DataFrame

Trains the model with train data and then performs predictions with the trained algorithm over the test data.

**Parameters**

- **data_train** (*Pandas Dataframe*) – Train data.
- **data_test** (*Pandas Dataframe*) – Test data.
- **labels_train** (*Pandas Dataframe*) – Train labels.
- **labels_test** (*Pandas Dataframe*) – Test labels.

**Returns**
Results of the predictions made on the test set.

**Return type**
Pandas Dataframe

**labels_per_sequence**(*labels: DataFrame*) → bool

Use only with time series datasets. Checks if the labels file of the chosen dataset has an index format with sequences only or sequences and time. *This method in future updates will be changed to a specific class for time series.*

**Parameters**
**labels** (*Pandas Dataframe*) – Labels file of the dataset.

**Returns**
True if the labels contains 1 level of index with sequence or False if the labels file contains 2 leves with sequence and time.

**Return type**
bool

**Raises**
**RuntimeError** – If the number of index levels is invalid.

**load_input**(*in_dir: str*) -> (<*class 'pandas.core.frame.DataFrame'*>, <*class 'pandas.core.frame.DataFrame'*>)

Read parquet data and labels files of the chosen dataset.

**Parameters**
**in_dir** (*str*) – Input directory where the files are located.

**Returns**

- **data** (*Pandas Dataframe*) – Loaded data file.
- **labels** (*Pandas Dataframe*) – Loaded labels file.

**save_output**(*results: DataFrame*, *out_dir: str*, *idx: int*) → str

Save algorithms output to parquet format.

**Parameters**

- **results** (*Pandas Dataframe*) – Dataframe with the results of the execution.
- **out_dir** (*str*) – Output directory where the results will be saved.
- **idx** (*int*) – Index of the algorithm in the Experiment. Userful in case you want to experiment with several algorithms.

**show_start_message**()

Simple method to print on the terminal the name of the algorithm to be executed.

### Methods

| | |
|---|---|
| *__init__*([encoding_dim, optimizer, loss_f, ...]) | |
| *execute*(idx, in_dir, out_dir) | Performs a complete execution of the algorithm, loading input data, performing a run through the folds and saving the results. |
| *find_anomaly_threshold*(values, n_features) | Finds anomaly threshold for threshold based algorithms. |
| *fit_predict*(data_train, data_test, ...) | Trains the model with train data and then performs predictions with the trained algorithm over the test data. |
| *labels_per_sequence*(labels) | Use only with time series datasets. |
| *load_input*(in_dir) | Read parquet data and labels files of the chosen dataset. |
| *save_output*(results, out_dir, idx) | Save algorithms output to parquet format. |
| *show_start_message*() | Simple method to print on the terminal the name of the algorithm to be executed. |

## conmo.algorithms.PretrainedRandomForest

**class** conmo.algorithms.**PretrainedRandomForest**(*pretrained: bool*, *max_depth: Optional[int] = None*, *random_state: Optional[int] = None*, *n_estimators: Optional[int] = None*, *path: Optional[str] = None*)

**__init__**(*pretrained: bool*, *max_depth: Optional[int] = None*, *random_state: Optional[int] = None*, *n_estimators: Optional[int] = None*, *path: Optional[str] = None*) → None

**execute**(*idx: int*, *in_dir: str*, *out_dir: str*) → str

Performs a complete execution of the algorithm, loading input data, performing a run through the folds and saving the results.

**Parameters**

- **idx** (*int*) – Index of the algorithm in the Experiment. Userful in case you want to experiment with several algorithms.
- **in_dir** (*str*) – Intermediate directory where the input data to the algorithm is stored.

- **out_dir** (*str*) – Intermediate directory where the output data (predictios of the algorithm) will be stored.

**Returns**
Name of the output directory.

**Return type**
str

**fit_predict**(*data_train: DataFrame*, *data_test: DataFrame*, *labels_train: DataFrame*, *labels_test: DataFrame*) → DataFrame

Trains the model with train data and then performs predictions with the trained algorithm over the test data.

**Parameters**

- **data_train** (*Pandas Dataframe*) – Train data.
- **data_test** (*Pandas Dataframe*) – Test data.
- **labels_train** (*Pandas Dataframe*) – Train labels.
- **labels_test** (*Pandas Dataframe*) – Test labels.

**Returns**
Results of the predictions made on the test set.

**Return type**
Pandas Dataframe

**labels_per_sequence**(*labels: DataFrame*) → bool

Use only with time series datasets. Checks if the labels file of the chosen dataset has an index format with sequences only or sequences and time. *This method in future updates will be changed to a specific class for time series.*

**Parameters**
**labels** (*Pandas Dataframe*) – Labels file of the dataset.

**Returns**
True if the labels contains 1 level of index with sequence or False if the labels file contains 2 leves with sequence and time.

**Return type**
bool

**Raises**
**RuntimeError** – If the number of index levels is invalid.

**load_input**(*in_dir: str) -> (<class 'pandas.core.frame.DataFrame'>*, *<class 'pandas.core.frame.DataFrame'>*)

Read parquet data and labels files of the chosen dataset.

**Parameters**
**in_dir** (*str*) – Input directory where the files are located.

**Returns**

- **data** (*Pandas Dataframe*) – Loaded data file.
- **labels** (*Pandas Dataframe*) – Loaded labels file.

**load_weights**()

Load pretrained model/weights for the algorithm's path.

**save_output**(*results: DataFrame*, *out_dir: str*, *idx: int*) → str

Save algorithms output to parquet format.

**Parameters**

- **results** (*Pandas Dataframe*) – Dataframe with the results of the execution.
- **out_dir** (*str*) – Output directory where the results will be saved.
- **idx** (*int*) – Index of the algorithm in the Experiment. Userful in case you want to experiment with several algorithms.

**show_start_message**()

Simple method to print on the terminal the name of the algorithm to be executed.

### Methods

| | |
|---|---|
| *__init__*(pretrained[, max_depth, ...]) | |
| *execute*(idx, in_dir, out_dir) | Performs a complete execution of the algorithm, loading input data, performing a run through the folds and saving the results. |
| *fit_predict*(data_train, data_test, ...) | Trains the model with train data and then performs predictions with the trained algorithm over the test data. |
| *labels_per_sequence*(labels) | Use only with time series datasets. |
| *load_input*(in_dir) | Read parquet data and labels files of the chosen dataset. |
| *load_weights*() | Load pretrained model/weights for the algorithm's path. |
| *save_output*(results, out_dir, idx) | Save algorithms output to parquet format. |
| *show_start_message*() | Simple method to print on the terminal the name of the algorithm to be executed. |

## conmo.algorithms.PretrainedMultilayerPerceptron

**class** conmo.algorithms.**PretrainedMultilayerPerceptron**(*pretrained: bool*, *input_len: int*, *random_seed: Optional[int] = None*, *path: Optional[str] = None*)

**__init__**(*pretrained: bool*, *input_len: int*, *random_seed: Optional[int] = None*, *path: Optional[str] = None*) → None

**build_mlp**() → Sequential

Auxiliary method for building the multilayer percetron.

**Returns**
**model** – Keras model built.

**Return type**
tf.keras.Model

**execute**(*idx: int*, *in_dir: str*, *out_dir: str*) → str

Performs a complete execution of the algorithm, loading input data, performing a run through the folds and saving the results.

**Parameters**

- **idx** (*int*) – Index of the algorithm in the Experiment. Userful in case you want to experiment with several algorithms.
- **in_dir** (*str*) – Intermediate directory where the input data to the algorithm is stored.
- **out_dir** (*str*) – Intermediate directory where the output data (predictios of the algorithm) will be stored.

**Returns**
Name of the output directory.

**Return type**
str

**fit_predict**(*data_train: DataFrame*, *data_test: DataFrame*, *labels_train: DataFrame*, *labels_test: DataFrame*) → DataFrame

Trains the model with train data and then performs predictions with the trained algorithm over the test data.

**Parameters**

- **data_train** (*Pandas Dataframe*) – Train data.
- **data_test** (*Pandas Dataframe*) – Test data.
- **labels_train** (*Pandas Dataframe*) – Train labels.
- **labels_test** (*Pandas Dataframe*) – Test labels.

**Returns**
Results of the predictions made on the test set.

**Return type**
Pandas Dataframe

**labels_per_sequence**(*labels: DataFrame*) → bool

Use only with time series datasets. Checks if the labels file of the chosen dataset has an index format with sequences only or sequences and time. *This method in future updates will be changed to a specific class for time series.*

**Parameters**
**labels** (*Pandas Dataframe*) – Labels file of the dataset.

**Returns**
True if the labels contains 1 level of index with sequence or False if the labels file contains 2 leves with sequence and time.

**Return type**
bool

**Raises**
**RuntimeError** – If the number of index levels is invalid.

**load_input**(*in_dir: str) -> (<class 'pandas.core.frame.DataFrame'>*, *<class 'pandas.core.frame.DataFrame'>*)

Read parquet data and labels files of the chosen dataset.

**Parameters**
**in_dir** (*str*) – Input directory where the files are located.

**Returns**

- **data** (*Pandas Dataframe*) – Loaded data file.
- **labels** (*Pandas Dataframe*) – Loaded labels file.

**load_weights**() → None

Load pretrained model/weights for the algorithm's path.

**save_output**(*results: DataFrame*, *out_dir: str*, *idx: int*) → str

Save algorithms output to parquet format.

**Parameters**

- **results** (*Pandas Dataframe*) – Dataframe with the results of the execution.
- **out_dir** (*str*) – Output directory where the results will be saved.
- **idx** (*int*) – Index of the algorithm in the Experiment. Userful in case you want to experiment with several algorithms.

**show_start_message**()

Simple method to print on the terminal the name of the algorithm to be executed.

### Methods

| | |
|---|---|
| *__init__*(pretrained, input_len[, ...]) | |
| *build_mlp*() | Auxiliary method for building the multilayer percetron. |
| *execute*(idx, in_dir, out_dir) | Performs a complete execution of the algorithm, loading input data, performing a run through the folds and saving the results. |
| *fit_predict*(data_train, data_test, ...) | Trains the model with train data and then performs predictions with the trained algorithm over the test data. |
| *labels_per_sequence*(labels) | Use only with time series datasets. |
| *load_input*(in_dir) | Read parquet data and labels files of the chosen dataset. |
| *load_weights*() | Load pretrained model/weights for the algorithm's path. |
| *save_output*(results, out_dir, idx) | Save algorithms output to parquet format. |
| *show_start_message*() | Simple method to print on the terminal the name of the algorithm to be executed. |

### conmo.algorithms.PretrainedCNN1D

class conmo.algorithms.**PretrainedCNN1D**(*pretrained: bool*, *input_len: int*, *random_seed: Optional[int] = None*, *path: Optional[str] = None*)

**__init__**(*pretrained: bool*, *input_len: int*, *random_seed: Optional[int] = None*, *path: Optional[str] = None*) → None

**build_cnn_1d**() → Sequential

Auxiliary method for building the 1D convolutional neural network.

**Returns**
**model** – Keras model built.

**Return type**
tf.keras.Model

**execute**(*idx: int*, *in_dir: str*, *out_dir: str*) → str

Performs a complete execution of the algorithm, loading input data, performing a run through the folds and saving the results.

**Parameters**

- **idx** (*int*) – Index of the algorithm in the Experiment. Userful in case you want to experiment with several algorithms.
- **in_dir** (*str*) – Intermediate directory where the input data to the algorithm is stored.
- **out_dir** (*str*) – Intermediate directory where the output data (predictios of the algorithm) will be stored.

**Returns**
Name of the output directory.

**Return type**
str

**fit_predict**(*data_train: DataFrame*, *data_test: DataFrame*, *labels_train: DataFrame*, *labels_test: DataFrame*) → DataFrame

Trains the model with train data and then performs predictions with the trained algorithm over the test data.

**Parameters**

- **data_train** (*Pandas Dataframe*) – Train data.
- **data_test** (*Pandas Dataframe*) – Test data.
- **labels_train** (*Pandas Dataframe*) – Train labels.
- **labels_test** (*Pandas Dataframe*) – Test labels.

**Returns**
Results of the predictions made on the test set.

**Return type**
Pandas Dataframe

**labels_per_sequence**(*labels: DataFrame*) → bool

Use only with time series datasets. Checks if the labels file of the chosen dataset has an index format with sequences only or sequences and time. *This method in future updates will be changed to a specific class for time series.*

**Parameters**
**labels** (*Pandas Dataframe*) – Labels file of the dataset.

**Returns**
True if the labels contains 1 level of index with sequence or False if the labels file contains 2 leves with sequence and time.

**Return type**
bool

**Raises**
**RuntimeError** – If the number of index levels is invalid.

**load_input**(*in_dir: str) -> (<class 'pandas.core.frame.DataFrame'>, <class 'pandas.core.frame.DataFrame'>*)

Read parquet data and labels files of the chosen dataset.

**Parameters**
**in_dir** (*str*) – Input directory where the files are located.

**Returns**

- **data** (*Pandas Dataframe*) – Loaded data file.
- **labels** (*Pandas Dataframe*) – Loaded labels file.

**load_weights**() → None

Load pretrained model/weights for the algorithm's path.

**save_output**(*results: DataFrame*, *out_dir: str*, *idx: int*) → str

Save algorithms output to parquet format.

**Parameters**

- **results** (*Pandas Dataframe*) – Dataframe with the results of the execution.
- **out_dir** (*str*) – Output directory where the results will be saved.
- **idx** (*int*) – Index of the algorithm in the Experiment. Userful in case you want to experiment with several algorithms.

**show_start_message**()

Simple method to print on the terminal the name of the algorithm to be executed.

#### Methods

| | |
|---|---|
| *__init__*(pretrained, input_len[, ...]) | |
| *build_cnn_1d*() | Auxiliary method for building the 1D convolutional neural network. |
| *execute*(idx, in_dir, out_dir) | Performs a complete execution of the algorithm, loading input data, performing a run through the folds and saving the results. |
| *fit_predict*(data_train, data_test, ...) | Trains the model with train data and then performs predictions with the trained algorithm over the test data. |
| *labels_per_sequence*(labels) | Use only with time series datasets. |
| *load_input*(in_dir) | Read parquet data and labels files of the chosen dataset. |
| *load_weights*() | Load pretrained model/weights for the algorithm's path. |
| *save_output*(results, out_dir, idx) | Save algorithms output to parquet format. |
| *show_start_message*() | Simple method to print on the terminal the name of the algorithm to be executed. |

### 4.1.6 conmo.metrics

The *conmo.metrics* submodule contains everything necessary to add new ways of measuring the effectiveness of the implemented algorithms. Accuracy and RMSPE are currently implemented.

| | |
|---|---|
| *metrics.metric.Metric*() | |
| *metrics.Accuracy*([normalize]) | |
| *metrics.RMSPE*([normalize]) | |

#### conmo.metrics.metric.Metric

**class** conmo.metrics.metric.**Metric**

**__init__**()

**abstract calculate**(*idx: int*, *algorithms: Iterable[str]*, *last_preprocess_dir: str*, *algorithms_dir: str*, *metrics_dir: str*) → None

Calculates specific metric for each of the algorithms' results.

**Parameters**

- **idx** (*str*) – Index of the metric in the Experiment. Userful in case you want to calculate several metrics.
- **algoritmss** (*Iterable[str]*) – List of names of the selected algorithms.
- **last_preprocess_dir** – Name of the directory where the ground truth is located

- **algorithms_dir** – Name of the directory where the results of the algorithms executions are stored.
- **metrics_dir** – Name of th edirectory where the results will be stored.

**labels_per_sequence**(*labels: DataFrame*) → bool

Use only with time series datasets. Checks if the labels file of the chosen dataset has an index format with sequences only or sequences and time. *This method in future updates will be changed to a specific class for time series.*

**Parameters**
**labels** (*Pandas Dataframe*) – Labels file of the dataset.

**Returns**
True if the labels contains 1 level of index with sequence or False if the labels file contains 2 leves with sequence and time.

**Return type**
bool

**Raises**
**RuntimeError** – If the number of index levels is invalid.

**load_results**(*algorithm: str*, *algorithms_dir: str*) → DataFrame

Load results for a specific algorthm.

**Parameters**

- **algoritm** (*str*) – Name of the selected algorithm.
- **algorithms_dir** (*str*) – Name of the directory where the results of the algorithms executions are stored.

**Returns**
Dataframe cantainig the results (predictions).

**Return type**
Pandas Dataframe

**load_truth**(*last_preprocess_dir: str*)

Load labels from the last preprocess directory.

**Parameters**
**last_preprocess_dir** (*str*) – Last diretory where the labels dataframe was stored.

**Returns**
Dataframe cantainig the labels.

**Return type**
Pandas Dataframe

**problem_label**(*truth: DataFrame*) → str

Determinates the nature of the problem by identifying the column's name of the labels.

**Parameters**
**truth** (*Pandas Dataframe*) – Labels file of the dataset.

**Returns**
Returns the column for the metric.

**Return type**
str

**Raises**

**RuntimeError** – If the labels of the ground truth are invalid for the problem.

**save_output**(*metric: DataFrame*, *idx: int*, *metrics_dir: str*) → None

Save metric's output to disk.

**Parameters**

- **metric** (*Pandas Dataframe*) – Dataframe containing the metric's results.
- **idx** (*int*) – Index of the metric in the Experiment. Userful in case you want to calculate several metrics.
- **metrics_dir** (*str*) – Name of the directory where the results will be stored.

**show_start_message**()

Simple method to print on the terminal the name of the used metric.

### Methods

| | |
|---|---|
| *__init__*() | |

| | |
|---|---|
| *calculate*(idx, algorithms, ...) | Calculates specific metric for each of the algorithms' results. |
| *labels_per_sequence*(labels) | Use only with time series datasets. |
| *load_results*(algorithm, algorithms_dir) | Load results for a specific algorthm. |
| *load_truth*(last_preprocess_dir) | Load labels from the last preprocess directory. |
| *problem_label*(truth) | Determinates the nature of the problem by identifying the column's name of the labels. |
| *save_output*(metric, idx, metrics_dir) | Save metric's output to disk. |
| *show_start_message*() | Simple method to print on the terminal the name of the used metric. |

## conmo.metrics.Accuracy

**class** conmo.metrics.**Accuracy**(*normalize: bool = True*)

**__init__**(*normalize: bool = True*) → None

**calculate**(*idx: int*, *algorithms: Iterable[str]*, *last_preprocess_dir: str*, *algorithms_dir: str*, *metrics_dir: str*) → None

Calculates specific metric for each of the algorithms' results.

**Parameters**

- **idx** (*str*) – Index of the metric in the Experiment. Userful in case you want to calculate several metrics.
- **algoritmss** (*Iterable[str]*) – List of names of the selected algorithms.
- **last_preprocess_dir** – Name of the directory where the ground truth is located

- **algorithms_dir** – Name of the directory where the results of the algorithms executions are stored.
- **metrics_dir** – Name of th edirectory where the results will be stored.

**labels_per_sequence**(*labels: DataFrame*) → bool

Use only with time series datasets. Checks if the labels file of the chosen dataset has an index format with sequences only or sequences and time. *This method in future updates will be changed to a specific class for time series.*

**Parameters**
**labels** (*Pandas Dataframe*) – Labels file of the dataset.

**Returns**
True if the labels contains 1 level of index with sequence or False if the labels file contains 2 leves with sequence and time.

**Return type**
bool

**Raises**
**RuntimeError** – If the number of index levels is invalid.

**load_results**(*algorithm: str*, *algorithms_dir: str*) → DataFrame

Load results for a specific algorthm.

**Parameters**

- **algoritm** (*str*) – Name of the selected algorithm.
- **algorithms_dir** (*str*) – Name of the directory where the results of the algorithms executions are stored.

**Returns**
Dataframe cantainig the results (predictions).

**Return type**
Pandas Dataframe

**load_truth**(*last_preprocess_dir: str*)

Load labels from the last preprocess directory.

**Parameters**
**last_preprocess_dir** (*str*) – Last diretory where the labels dataframe was stored.

**Returns**
Dataframe cantainig the labels.

**Return type**
Pandas Dataframe

**problem_label**(*truth: DataFrame*) → str

Determinates the nature of the problem by identifying the column's name of the labels.

**Parameters**
**truth** (*Pandas Dataframe*) – Labels file of the dataset.

**Returns**
Returns the column for the metric.

**Return type**
str

**Raises**

**RuntimeError** – If the labels of the ground truth are invalid for the problem.

**save_output**(*metric: DataFrame*, *idx: int*, *metrics_dir: str*) → None

Save metric's output to disk.

**Parameters**

- **metric** (*Pandas Dataframe*) – Dataframe containing the metric's results.
- **idx** (*int*) – Index of the metric in the Experiment. Userful in case you want to calculate several metrics.
- **metrics_dir** (*str*) – Name of the directory where the results will be stored.

**show_start_message**()

Simple method to print on the terminal the name of the used metric.

### Methods

| | |
|---|---|
| *__init__*([normalize]) | |

| | |
|---|---|
| *calculate*(idx, algorithms, ...) | Calculates specific metric for each of the algorithms' results. |
| *labels_per_sequence*(labels) | Use only with time series datasets. |
| *load_results*(algorithm, algorithms_dir) | Load results for a specific algorthm. |
| *load_truth*(last_preprocess_dir) | Load labels from the last preprocess directory. |
| *problem_label*(truth) | Determinates the nature of the problem by identifying the column's name of the labels. |
| *save_output*(metric, idx, metrics_dir) | Save metric's output to disk. |
| *show_start_message*() | Simple method to print on the terminal the name of the used metric. |

## conmo.metrics.RMSPE

**class** conmo.metrics.**RMSPE**(*normalize: bool = True*)

**__init__**(*normalize: bool = True*) → None

**calculate**(*idx: int*, *algorithms: Iterable[str]*, *last_preprocess_dir: str*, *algorithms_dir: str*, *metrics_dir: str*) → None

Calculates specific metric for each of the algorithms' results.

**Parameters**

- **idx** (*str*) – Index of the metric in the Experiment. Userful in case you want to calculate several metrics.
- **algoritmss** (*Iterable[str]*) – List of names of the selected algorithms.
- **last_preprocess_dir** – Name of the directory where the ground truth is located

- **algorithms_dir** – Name of the directory where the results of the algorithms executions are stored.
- **metrics_dir** – Name of th edirectory where the results will be stored.

**labels_per_sequence**(*labels: DataFrame*) → bool

Use only with time series datasets. Checks if the labels file of the chosen dataset has an index format with sequences only or sequences and time. *This method in future updates will be changed to a specific class for time series.*

**Parameters**
**labels** (*Pandas Dataframe*) – Labels file of the dataset.

**Returns**
True if the labels contains 1 level of index with sequence or False if the labels file contains 2 leves with sequence and time.

**Return type**
bool

**Raises**
**RuntimeError** – If the number of index levels is invalid.

**load_results**(*algorithm: str*, *algorithms_dir: str*) → DataFrame

Load results for a specific algorthm.

**Parameters**

- **algoritm** (*str*) – Name of the selected algorithm.
- **algorithms_dir** (*str*) – Name of the directory where the results of the algorithms executions are stored.

**Returns**
Dataframe cantainig the results (predictions).

**Return type**
Pandas Dataframe

**load_truth**(*last_preprocess_dir: str*)

Load labels from the last preprocess directory.

**Parameters**
**last_preprocess_dir** (*str*) – Last diretory where the labels dataframe was stored.

**Returns**
Dataframe cantainig the labels.

**Return type**
Pandas Dataframe

**problem_label**(*truth: DataFrame*) → str

Determinates the nature of the problem by identifying the column's name of the labels.

**Parameters**
**truth** (*Pandas Dataframe*) – Labels file of the dataset.

**Returns**
Returns the column for the metric.

**Return type**
str

**Raises**

**RuntimeError** – If the labels of the ground truth are invalid for the problem.

**rmspe**(*y_true: ndarray*, *y_pred: ndarray*) → ndarray

Compute Root Mean Square Percentage Error between two arrays.

**save_output**(*metric: DataFrame*, *idx: int*, *metrics_dir: str*) → None

Save metric's output to disk.

**Parameters**

- **metric** (*Pandas Dataframe*) – Dataframe containing the metric's results.
- **idx** (*int*) – Index of the metric in the Experiment. Userful in case you want to calculate several metrics.
- **metrics_dir** (*str*) – Name of the directory where the results will be stored.

**show_start_message**()

Simple method to print on the terminal the name of the used metric.

### Methods

| | |
|---|---|
| *__init__*([normalize]) | |

| | |
|---|---|
| *calculate*(idx, algorithms, ...) | Calculates specific metric for each of the algorithms' results. |
| *labels_per_sequence*(labels) | Use only with time series datasets. |
| *load_results*(algorithm, algorithms_dir) | Load results for a specific algorthm. |
| *load_truth*(last_preprocess_dir) | Load labels from the last preprocess directory. |
| *problem_label*(truth) | Determinates the nature of the problem by identifying the column's name of the labels. |
| *rmspe*(y_true, y_pred) | Compute Root Mean Square Percentage Error between two arrays. |
| *save_output*(metric, idx, metrics_dir) | Save metric's output to disk. |
| *show_start_message*() | Simple method to print on the terminal the name of the used metric. |

CHAPTER

FIVE

# DEVELOPMENT GUIDE

## 5.1 Development guide

### 5.1.1 Possibilities of Conmo

Conmo framework has been designed to be user-friendly for recreating and evaluating experiments, but also for adding new algorithms, datasets, preprocesses, etc. This section explains the possibilities offered by this framework when implementing new submodules. We believe that using and contributing to Conmo can benefit all types of users, as well as helping to standardise comparisons between results from different scientific articles.

If you still have doubts about the implementation of new components to the framework, you can take a look at the API reference, examples or contact the developers.

### 5.1.2 Add a new dataset

*Dataset* is the core abstract class for every dataset in Conmo and contains basic methods and attributes that are common for all datasets. At the same time, two classes depend on it and differ acording to where the original data is stored:

- ***LocalDataset*:**
  Is the abstract class in charge of handling datasets that are stored locally on the computer where Conmo will be running. The main method of this class is *LocalDataset.load()*. It's in charge of parsing the original dataset files to Conmo's format and moving them to the data folder. It's an abstract method wich needs to be implemented in every local dataset. There is also an abstract method `feed_pipeline()` to copy selected data to pipeline step folder.
- ***RemoteDataset*:**
  In case the dataset to be implemented is originally located on a web server, a Git repository or other remote hosting, the RemoteDataset class is available in Conmo. Among all its methos, it's remarkable the *RemoteDataset.download()* to download the dataset from a remote URL.

For adding a new local dataset to the framework you need to create a new class that inherits from *LocalDataset* and override the following methods:

- ***LocalDataset.__init__()*:**
  This is the constructor of the class. Here you can call the constructor of the father class to assign the path to thw original dataset. Here you can also define some attributes of the class, like the label's columns names, feature's names. Also you can assign the subdataset that you want to instanciate.
- ***LocalDataset.dataset_files()*:**
  This method must return a list with all the files (data and labels) that compounds the dataset.

- *LocalDataset.load()*:
  This method must convert all raw dataset files to the appropriate format for Conmo's pipeline. For each of the datasets, first read and load the data and labels into Pandas dataframes, then concatenate them (e.g. train data and test data will be concatenated in one dataframe, the same for test) and finally save them in parquet format. Some considerations to take into account:
  - Data and labels dataframes will have at least a multi-index for sequence and time. You can consult more information in the Pandas documentation.
  - The columns index must start at 1.
  - If there dataset is only splittered into train and test, then there will be 2 sequences, one per set.
  - In case the dataset is a time series with sequences, train sequences go after the test sequences.
- *LocalDataset.feed_pipeline()*:
  This method is used to copy the dataset from *data* directory to the directory *Dataset* of the experiment.
- **LocalDataset.sklearn_predefined_split()**:
  If you plan to use the Predefined Split from the Sklearn library your class must implement this method. It must generate an array of indexes of same length as sequences to be used with PredefinedSplit. The index must start at 0.

For adding a new remote dataset to the framework the procedure is almost identical to a local dataset. You need to create a new class that inherits from *RemoteDataset* and override the following methods:

- *RemoteDataset.__init__()*:
  This is the constructor of the class. Here you can call the constructor of the father class to assign the path to thw original dataset. You can also define some attributes of the class, like the label's columns names, features's names. , file format, URL and checksum. Also you can assign the subdataset that you want to instanciate.
- *RemoteDataset.dataset_files()*:
  This method must return a list with all the files (data and labels) that compounds the dataset.
- *RemoteDataset.parse_to_package()*:
  Almost identical to *LocalDataset.load()*.
- *RemoteDataset.feed_pipeline()*:
  This method is used to pass the dataset from *data* directory to the directory *Dataset* of the experiment.
- **RemoteDataset.sklearn_predefined_split()**:
  If you plan to use the Predefined Split from the Sklearn library your class must implement this method. It must generate an array of indexes of same length as sequences to be used with PredefinedSplit. The index must start at 0.

### 5.1.3 Add a new algorithm

Conmo provides a core abstract class named *Algorithm* that contains the basic methods for the operation of any algorithm, mainly training with a training set, performing a prediction over test, loading and saving input and output data. Depending on the type of anomaly detection algorithm to be implemented, there are two classes depending on the operation of the method:

- *AnomalyDetectionThresholdBasedAlgorithm*:
  If your algorithm needs to calculate a threshold to determine which samples are anomalous it must inherit from this class. For example: PCA Mahalanobis.
- *AnomalyDetectionClassBasedAlgorithm*:
  If your algorithm identifies by classes the normal sequences from the anomalous ones, it must inherit from this class. For example: One Class SVM.

- **PretrainedAlgorithm:**
  Check out this class if your algorithm was pre-trained prior to running an experiment, i.e. it is not necessary to train it during the experiment. It is required to be able to define the path where the pre-trained model is stored on disk.

For adding a new algorithm to the framework you need to create a new class that inherits from one of these classes depending of the type of the algorithm and override the following methods:

- **__init__():**
  Constructor of the class. Here you can initialize all the hyperparameters needed for the algorithm. Also you can fix random seeds of Tensorflow, Numpy, etc here for reproducibilty purposes.
- **fit_predict():**
  Method responsible of building, training the model with the training data and testing it with the test set. In case your algorithm is threshold-based, it will be necessary to verify that each output in the test set exceeds that threshold to determine that it is anomalous. In the case of a class-based algorithm, depending on the output, it will be necessary to identify whether it is an outlier or an anomaly. Finally, the output dataframe has to be generated with the labels by sequence or by time.
- **find_anomaly_threshold():**
  In case the algorithm is threshold based, the threshold selection can be customised overriding this method.

You can add auxiliary methods for model construction, weights loading, etc. in case the model structure is very complex.

### 5.1.4 Add a new splitter

The core abstract class is *Splitter* and provides some methods to load inputs, save outputs and check it the input was already splittered. For adding a new splitter you must create a new class that inherits from *Splitter* and implements the method `Transform()`. If the splitters you want to implement is available on Scikit-Learn library, we provide the class `SklearnSplitter` and indicating the name of the splitter to be used will allow you to use it in your experiment.

### 5.1.5 Add a new preprocess

`ExtendedPreprocess` Class is used for the implementation of new preprocessings in the pipeline. `ExtendedPreprocess` inherits from the core abstract class `Preprocess` and provides a constructor in order to define which parts of the dataset will be modified by the preprocessing: labels, data, test or train. Also permits to apply the preprocess to a specyfic set of columns. To define a new preprocess you only need to create a new class than inherits from `ExtendedPreprocess` and implements the method `Transform()`, where the preprocessing will be applied to th datset. If the preprocess you want to implement is available on Sklearn library, we provide the class `SklearnPreprocess` and indicating the name of the preprocessing to be used will allow you to use it in your experiment. In order to make things easier, the `CustomPreprocess` class is available to implement a preprocessing tool from a function, which will be passed as an argument in the constructor. For additional information you can have a look at the example *nasa_cmapps.py*.

### 5.1.6 Add a new metric

You can add a new metric by creating a new class that inherits from the abstract class *Metric*.

The only method you have to take care is:

- **calculate():**
  Based on the outputs of the algorithms and the number of folds, the results are computed and the metrics dataframe is created and stored.

### 5.1.7 Coding conventions

The following tools are used to ensure that new software being added to Conmo meets minimum quality and format requirements:

- **Autopep8**: We use this tool to automatically format our Python code to conform to the PEP 8 style guide. It uses the pycodestyle utility to determine what parts of the code needs to be formatted.
- **Isort**: We use this library to sort imports alphabetically, and automatically separated into sections and by type.
- **Pytest**: To ensure that the output format of a new designed step (algorithm, dataset, etc) is correct we use Pytest framework to testing the new code. This tetsing frameworks is easy to use and supoort complex testing at the same. At the moment we are finishing the implementation of tests on the existing code, so there could be parts that may be modified in future updates.

CHAPTER

SIX

# KNOWN ISSUES & LIMITATIONS

## 6.1 Known Issues & Limitations

We are aware that Conmo is still in a very early stage of development, so it is likely that as its use increases, various bugs will appear. Bugs that are detected will be published on this page in order to make it easier for users to prevent them. However, the Conmo development team is actively looking for and fixing any detected bugs. Please, if you find a bug/issue that does not appear on this list, we would be grateful if you could email us at mym.inv.uniovi@gmail.com or post an issue on our Github. Thanks in advance.

| Issue ID | Severity | Description |
|---|---|---|
| 001_split | Low | There are some problems with the use of Scikit-Learn's Time Series Splitter in the experiments. We are working on resolving them. |
| 002_rul | Medium | rul_rve.py example seems to be failing during the metric calculation step. |

CHAPTER

SEVEN

# FREQUENTLY ASKED QUESTIONS

## 7.1 Frequently Asked Questions

### 7.1.1 How can I contribute to Conmo?

Depending on your profile and your intended use, you can contribute in different ways: The simplest way to contribute to Conmo is to use it to reproduce some experiments and then cite it. However, you can also contribute by implementing new algorithms, datasets, etc. that can then be used by everyone to perform experiments. Finally, reporting bugs in the functioning of Conmo can also be considered a way to collaborate with the project.

### 7.1.2 I don't have a great knowledge of programming, can I still use Conmo?

Conmo intends to focus on all types of scientists, regardless of their specialisation. Generally speaking, we can distinguish two types of people who will use Conmo:

1. People who only want to reproduce experiments that are already integrated only need basic programming knowledge, since Python is a simple programming language and the complexity has been largely encapsulated.
2. People who want to collaborate by adding new algorithms, datasets, etc. need more in-depth programming knowledge. In particular Python language and object-oriented programming. However, the Conmo development team is actively looking for a way to simplify this kind of actions.

CHAPTER

EIGHT

# RELEASE NOTES

## 8.1 Release Notes

### 8.1.1 Conmo 1.0.1

Small documentation and CI errors fixed.

### 8.1.2 Conmo 1.0.0

First publicly available version.

# PYTHON MODULE INDEX

## C

# INDEX

## Symbols



## A



## B

## C



## D



## E

## F



## G



## I



## K



## L

## M



## N



## O

## P



## R



## S

## T