A2PM

Release 1.2.0

João Vitorino

Jun 29, 2023

INTRODUCTION

Welcome to the official documentation of the Adaptative Perturbation Pattern Method.

A2PM is the outcome of R&D activities carried out at:
Research Group on Intelligent Engineering and Computing for Advanced Innovation
and Development (GECAD), School of Engineering, Polytechnic of Porto (ISEP/IPP),
4249-015 Porto, Portugal.

CHAPTER

ONE

ADAPTATIVE PERTURBATION PATTERN METHOD

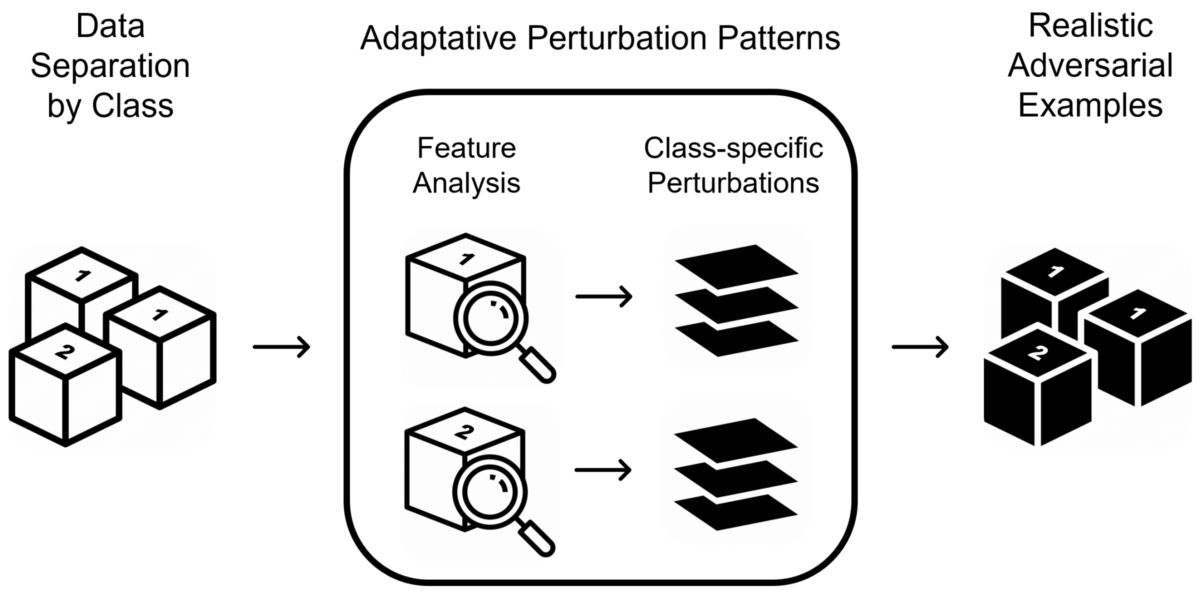
A2PM is a gray-box method for the generation of realistic adversarial examples. It benefits from a modular architecture to assign an independent sequence of adaptative perturbation patterns to each class, which analyze specific feature subsets to create valid and coherent data perturbations.

This method was developed to address the diverse constraints of domains with tabular data, such as cybersecurity. It can be advantageous for adversarial attacks against machine learning classifiers, as well as for adversarial training strategies. This Python 3 implementation provides out-of-the-box compatibility with the Scikit-Learn library.

If you use A2PM, please cite the primary research article: https://doi.org/10.3390/fi14040108

Check out the official documentation: https://a2pm.readthedocs.io/en/latest

Explore the public source code repository: https://github.com/vitorinojoao/a2pm


Data Separation by Class
Adaptative Perturbation Patterns
Feature Analysis
Class-specific Perturbations
Realistic Adversarial Examples
1
1
2
1
2
1
1
2

1.1 How To Install

The package and its dependencies can be installed using the pip package manager:

```
pip install a2pm
```

Alternatively, the repository can be downloaded and the package installed from the local directory:

```
pip install .
```

1.2 How To Setup

The package can be accessed through the following imports:

```
from a2pm import A2PMethod
from a2pm.callbacks import BaseCallback, MetricCallback, TimeCallback
from a2pm.patterns import BasePattern, CombinationPattern, IntervalPattern
from a2pm.wrappers import BaseWrapper, KerasWrapper, SklearnWrapper, TorchWrapper
```

A2PM can be created with a simple base configuration of Interval and/or Combination pattern sequences, which have several possible parameters:

```
pattern = (

    # First pattern to be applied: Interval
    {
        "type": "interval",
        "features": list(range(0, 20)),
        "integer_features": list(range(10, 20)),
        "ratio": 0.1,
        "max_ratio": 0.3,
        "missing_value": 0.0,
        "probability": 0.6,
    },

    # Second pattern to be applied: Combination
    {
        "type": "combination",
        "features": list(range(20, 40)),
        "locked_features": list(range(30, 40)),
        "probability": 0.4,
    },
)

method = A2PMethod(pattern)
```

To support domains with complex constraints, the method is highly configurable:

```
# General pattern sequence that will be applied to new data classes
pattern = (

```

(continues on next page)

(continued from previous page)

```
    # An instantiated pattern
    MyCustomPattern(1, 2),

    # A pattern configuration
    {
        "type": MyCustomPattern,
        "param_name_1": 3,
        "param_name_2": 4,
    },
)

# Pre-assigned mapping of data classes to pattern sequences
preassigned_patterns = {

    # None to disable the perturbation of this class
    "class_label_1": None,

    # Specific pattern sequence that will be applied to this class
    "class_label_2": (
        MyCustomPattern(5, 6),
        {
            "type": MyCustomPattern,
            "param_name_1": 7,
            "param_name_2": 8,
        },
    ),
}

method = A2PMethod(pattern, preassigned_patterns)
```

1.3 How To Use

A2PM can be utilized through the ‘fit’, ‘partial_fit’, ‘transform’ and ‘generate’ methods, as well as their respective shortcuts:

```
# Adapts to new data, and then creates adversarial examples
X_adversarial = method.fit_transform(X, y)

# Encapsulates a Tensorflow/Keras classification model
classifier = KerasWrapper(my_model, my_custom_class_labels)

# Adapts to new data, and then performs an untargeted attack against a classifier
X_adversarial = method.fit_generate(classifier, X, y)

# Adapts to new data, and then performs a targeted attack against a classifier
X_adversarial = method.fit_generate(classifier, X, y, y_target)
```

To analyze specific aspects of the method, callback functions can be called before the attack starts (iteration 0) and after each attack iteration (iteration 1, 2, . . .):

```python
X_adversarial = method.fit_generate(
    classifier,
    X,
    y,
    y_target,

    # Additional configuration
    iterations=10,
    patience=2,

    callback=[

        # Time consumption
        TimeCallback(verbose=2),

        # Evaluation metrics
        MetricCallback(classifier, y, my_custom_scorers, verbose=2),

        # An instantiated callback
        MyCustomCallback(),

        # A simple callback function
        MyCustomFunction,
    ],
)
```

CHAPTER

TWO

A2PMETHOD

class a2pm.**A2PMethod**(*pattern*, *preassigned_patterns=None*, *class_discriminator=<function A2PMethod.<lambda>>*, *seed=None*)

Bases: `sklearn.base.BaseEstimator`

Adaptative Perturbation Pattern Method.

A2PM generates realistic adversarial examples by assigning an independent sequence of adaptative patterns to each class, which analyze specific feature subsets to create valid and coherent data perturbations.

Note: Class-specific data perturbations can only be created if the class of each sample is identified, either as a label or a numeric representation. To obtain external Class IDs for internal use by this method, there are two alternatives:

- Specify a *class_discriminator* function;
- Provide the *y* parameter to the *fit*, *partial_fit*, *transform* and *generate* methods.

Parameters

- **pattern** (*pattern, config or tuple of patterns/configs*) – Default pattern (or pattern tuple) to be adapted for each new found class. Supports configurations to create patterns, as well as pre-fitted pattern instances.
- **preassigned_patterns** (*dict of 'Class ID - pattern' pairs (default None)*) – Pre-assigned mapping of specific classes to their specific patterns (or pattern tuples). Also supports configurations to create patterns, as well as pre-fitted pattern instances.

  { Class ID : pattern, Class ID : (pattern, pattern), Class ID : None }

  Preassign None to a Class ID to disable perturbations of that class.

  Set to None to disable pre-assignments, treating all classes as new.
- **class_discriminator** (*callable or None (default lambda)*) – Function to be used to identify the Class ID of each sample of input data *X*, in order to use class-specific patterns.

  class_discriminator(X) -> y

  If no discriminator is specified and the *y* parameter is not provided to a method, all samples will be assigned to the same general class. To prevent overlapping with regular Class IDs, that class has the *-2* ID. Therefore, the default function is:

  lambda X: numpy.full(X.shape[0], -2)

  Set to None to disable the default function, imposing the use of the *y* parameter for all methods.
- **seed** (*int, None or a generator (default None)*) – Seed for reproducible random number generation. If provided:

- For pattern configurations, it will override any configured seed;
- For already created patterns, it will not have any effect.

Variables

- `classes` (*`list of Class IDs`*) – The currently known classes. Only available after a call to *fit* or *partial_fit*.
- `class_mapping` (*`dict of 'Class ID - pattern' pairs`*) – The current mapping of known classes to their respective pattern tuples. Only available after a call to *fit* or *partial_fit*.

`fit`(*X*, *y=None*)

Fully adapts the method to new data.

First, the method is reset to the *preassigned_patterns*, be it configurations or pre-fitted pattern instances. Then, for new found classes, the default pattern will be assigned and updated. For classes with pre-assigned patterns, these will be updated.

Parameters

- **X** (*array-like in the (n_samples, n_features) shape*) – Input data.
- **y** (*array-like in the (n_samples,) shape or None (default None)*) – Class IDs of input data, to use class-specific patterns.

  Set to None to use the *class_discriminator* function.

Returns This A2PMethod instance.

Return type self

`partial_fit`(*X*, *y=None*)

Partially adapts the method to new data.

For new found classes, the default pattern will be assigned and updated. For known classes, either pre-assigned or previously found, their patterns will be updated.

Parameters

- **X** (*array-like in the (n_samples, n_features) shape*) – Input data.
- **y** (*array-like in the (n_samples,) shape or None (default None)*) – Class IDs of input data, to use class-specific patterns.

  Set to None to use the *class_discriminator* function.

Returns This A2PMethod instance.

Return type self

`transform`(*X*, *y=None*, *quantity=1*, *keep_original=False*) → numpy.ndarray

Applies the method to create adversarial examples.

Parameters

- **X** (*array-like in the (n_samples, n_features) shape*) – Input data.
- **y** (*array-like in the (n_samples,) shape or None (default None)*) – Class IDs of input data, to use class-specific patterns.

  Set to None to use the *class_discriminator* function.

- **quantity** (*int, > 0 (default 1)*) – Number of examples to create for each sample.

- **keep_original** (*bool (default False)*) – Signal to keep the original input data in the returned array, in addition to the created examples.

Returns

X_adversarial – Adversarial data, in the same order as input data.

If quantity > 1, the resulting array will be tiled:

example1_of_sample1

example1_of_sample2

example1_of_sample3

example2_of_sample1

example2_of_sample2

example2_of_sample3

…

If *keep_original* is signalled, the resulting array will contain the original input data and also be tiled:

sample1

sample2

sample3

example1_of_sample1

example1_of_sample2

example1_of_sample3

…

Return type numpy array of shape (n_samples * quantity, n_features)

`fit_transform`(*X*, *y=None*, *quantity=1*, *keep_original=False*) → numpy.ndarray

Fully adapts the method to new data, and then applies it to create adversarial examples.

First, the method is reset to the *preassigned_patterns*, be it configurations or pre-fitted pattern instances. Then, for new found classes, the default pattern will be assigned and updated. For classes with pre-assigned patterns, these will be updated.

Parameters

- **X** (*array-like in the (n_samples, n_features) shape*) – Input data.
- **y** (*array-like in the (n_samples,) shape or None (default None)*) – Class IDs of input data, to use class-specific patterns.

  Set to None to use the *class_discriminator* function.
- **quantity** (*int, > 0 (default 1)*) – Number of examples to create for each sample.
- **keep_original** (*bool (default False)*) – Signal to keep the original input data in the returned array, in addition to the created examples.

Returns

X_adversarial – Adversarial data, in the same order as input data.

If quantity > 1, the resulting array will be tiled.

If *keep_original* is signalled, the resulting array will contain the original input data and also be tiled.

Return type numpy array of shape (n_samples * quantity, n_features)

`partial_fit_transform`(*X*, *y=None*, *quantity=1*, *keep_original=False*) → numpy.ndarray

Partially adapts the method to new data, and then applies it to create adversarial examples.

For new found classes, the default pattern will be assigned and updated. For known classes, either pre-assigned or previously found, their patterns will be updated.

Parameters

- **X** (*array-like in the (n_samples, n_features) shape*) – Input data.
- **y** (*array-like in the (n_samples,) shape or None (default None)*) – Class IDs of input data, to use class-specific patterns.

  Set to None to use the *class_discriminator* function.
- **quantity** (*int, > 0 (default 1)*) – Number of examples to create for each sample.
- **keep_original** (*bool (default False)*) – Signal to keep the original input data in the returned array, in addition to the created examples.

Returns

X_adversarial – Adversarial data, in the same order as input data.

If quantity > 1, the resulting array will be tiled.

If *keep_original* is signalled, the resulting array will contain the original input data and also be tiled.

Return type numpy array of shape (n_samples * quantity, n_features)

`generate`(*classifier*, *X*, *y=None*, *y_target=None*, *iterations=10*, *patience=2*, *callback=None*) → numpy.ndarray

Applies the method to perform adversarial attacks against a classifier.

An attack can be untargeted, causing any misclassification, or targeted, seeking to reach a specific class. To perform a targeted attack, the class that should be reached for each sample must be provided in the *y_target* parameter.

Note: The misclassifications are caused on the class predictions of the classifier. These predictions are independent from the Class IDs provided in *y* or by the *class_discriminator* function, which remain for internal use only.

Parameters

- **classifier** (object with a *predict* method) – Fitted classifier to be attacked.
- **X** (*array-like in the (n_samples, n_features) shape*) – Input data.
- **y** (*array-like in the (n_samples,) shape or None (default None)*) – Class IDs of input data, to use class-specific patterns.

  Set to None to use the *class_discriminator* function.
- **y_target** (*array-like in the (n_samples,) shape or None (default None)*) – Class predictions that should be reached in a targeted attack.

  Set to None to perform an untargeted attack.
- **iterations** (*int, > 0 (default 10)*) – Maximum number of iterations that can be performed before ending the attack.

- **patience** (*int, >= 0 (default 2)*) – Patience for early stopping. Corresponds to the number of iterations without further misclassifications that can be performed before ending the attack.

  Set to 0 to disable early stopping.

- **callback** (*callable or list of callables*) – List of functions to be called before the attack starts (iteration 0), and after each attack iteration (iteration 1, 2, . . .).

  *callback(**kwargs)*

  callback(X, iteration, samples_left, samples_misclassified, nanoseconds)

  It can receive five parameters:

  – the current data (input data at iteration 0, and then adversarial data);

  – the current attack iteration;

  – the number of samples left to be misclassified;

  – the number of samples misclassified in the current iteration;

  – the number of nanoseconds consumed in the current iteration.

  For example, a simple function to print each iteration can be:

  *def callback(**kwargs): print(kwargs["iteration"])*

Returns **X_adversarial** – Adversarial data, in the same order as input data.

Return type numpy array of shape (n_samples, n_features)

`fit_generate`(*classifier*, *X*, *y=None*, *y_target=None*, *iterations=10*, *patience=2*, *callback=None*) → numpy.ndarray

Fully adapts the method to new data, and then applies it to perform adversarial attacks against a classifier.

First, the method is reset to the *preassigned_patterns*, be it configurations or pre-fitted pattern instances. Then, for new found classes, the default pattern will be assigned and updated. For classes with pre-assigned patterns, these will be updated.

An attack can be untargeted, causing any misclassification, or targeted, seeking to reach a specific class. To perform a targeted attack, the class that should be reached for each sample must be provided in the *y_target* parameter.

Note: The misclassifications are caused on the class predictions of the classifier. These predictions are independent from the Class IDs provided in *y* or by the *class_discriminator* function, which remain for internal use only.

Parameters

- **classifier** (object with a *predict* method) – Fitted classifier to be attacked.
- **X** (*array-like in the (n_samples, n_features) shape*) – Input data.
- **y** (*array-like in the (n_samples,) shape or None (default None)*) – Class IDs of input data, to use class-specific patterns.

  Set to None to use the *class_discriminator* function.

- **y_target** (*array-like in the (n_samples,) shape or None (default None)*) – Class predictions that should be reached in a targeted attack.

  Set to None to perform an untargeted attack.

- **iterations** (*int, > 0 (default 10)*) – Maximum number of iterations that can be performed before ending the attack.

- **patience** (*int, >= 0 (default 2)*) – Patience for early stopping. Corresponds to the number of iterations without further misclassifications that can be performed before ending the attack.

  Set to 0 to disable early stopping.

- **callback** (*callable or list of callables*) – List of functions to be called before the attack starts (iteration 0), and after each attack iteration (iteration 1, 2, . . .).

  *callback(**kwargs)*

  callback(X, iteration, samples_left, samples_misclassified, nanoseconds)

Returns **X_adversarial** – Adversarial data, in the same order as input data.

Return type numpy array of shape (n_samples, n_features)

`partial_fit_generate`(*classifier*, *X*, *y=None*, *y_target=None*, *iterations=10*, *patience=2*, *callback=None*) → numpy.ndarray

Partially adapts the method to new data, and then applies it to perform adversarial attacks against a classifier.

For new found classes, the default pattern will be assigned and updated. For known classes, either pre-assigned or previously found, their patterns will be updated.

An attack can be untargeted, causing any misclassification, or targeted, seeking to reach a specific class. To perform a targeted attack, the class that should be reached for each sample must be provided in the *y_target* parameter.

Note: The misclassifications are caused on the class predictions of the classifier. These predictions are independent from the Class IDs provided in *y* or by the *class_discriminator* function, which remain for internal use only.

Parameters

- **classifier** (object with a *predict* method) – Fitted classifier to be attacked.
- **X** (*array-like in the (n_samples, n_features) shape*) – Input data.
- **y** (*array-like in the (n_samples,) shape or None (default None)*) – Class IDs of input data, to use class-specific patterns.

  Set to None to use the *class_discriminator* function.

- **y_target** (*array-like in the (n_samples,) shape or None (default None)*) – Class predictions that should be reached in a targeted attack.

  Set to None to perform an untargeted attack.

- **iterations** (*int, > 0 (default 10)*) – Maximum number of iterations that can be performed before ending the attack.
- **patience** (*int, >= 0 (default 2)*) – Patience for early stopping. Corresponds to the number of iterations without further misclassifications that can be performed before ending the attack.

  Set to 0 to disable early stopping.

- **callback** (*callable or list of callables*) – List of functions to be called before the attack starts (iteration 0), and after each attack iteration (iteration 1, 2, . . .).

  *callback(**kwargs)*

  callback(X, iteration, samples_left, samples_misclassified, nanoseconds)

Returns **X_adversarial** – Adversarial data, in the same order as input data.

Return type numpy array of shape (n_samples, n_features)

CHAPTER

THREE

CALLBACKS

3.1 BaseCallback

class `a2pm.callbacks.`**`BaseCallback`**(*verbose=0*)

Bases: `object`

Base Attack Callback.

A callback records and/or prints specific values of each attack iteration of the *generate* method. This base class cannot be directly utilized.

It must be either a function or a class implementing the *__call__* method, according to one of the following signatures:

*__call__(self, **kwargs)*

__call__(self, X, iteration, samples_left, samples_misclassified, nanoseconds)

It can receive five parameters:

- the current data (input data at iteration 0, and then adversarial data);
- the current attack iteration;
- the number of samples left to be misclassified;
- the number of samples misclassified in the current iteration;
- the number of nanoseconds consumed in the current iteration.

For example, a simple function to print each iteration can be:

*def callback(**kwargs): print(kwargs["iteration"])*

Parameters **verbose** (*int, in {0, 1, 2} (default 0)*) – Verbosity level of the callback.

Set to 2 to enable a complete printing of the values and their descriptions, to 1 to enable a simple printing of the values, or to 0 to disable verbosity.

Variables **`values`** (*`list of values`*) – The values recorded at each iteration by an inheriting class. Empty list before that class is called.

3.2 MetricCallback

`class a2pm.callbacks.MetricCallback`(*classifier*, *y*, *scorers=[('Macro-averaged F1-Score', 'f1_macro')]*, *verbose=0*)

Bases: `a2pm.callbacks.base_callback.BaseCallback`

Metric Attack Callback.

Records the score of one or more metrics at each iteration.

The metrics are measured according to their respective scorer functions.

Parameters

- **classifier** (object with a *predict* method) – Fitted classifier to be evaluated, which should be the same classifier being attacked.
- **y** (*array-like in the (n_samples,) shape or None (default None)*) – Ground truth classes that the classifier should predict.
- **scorers** (*list of tuples of 'description, scorer'*) – Tuples of custom metric descriptions and respective scorer functions.

  Besides an actual scorer function, a Scikit-learn compatible description is also supported.

  The default scorer is the following:

  ("Macro-averaged F1-Score", "f1_macro")
- **verbose** (*int, in {0, 1, 2} (default 0)*) – Verbosity level of the callback.

  Set to 2 to enable a complete printing of the values and their descriptions, to 1 to enable a simple printing of the values, or to 0 to disable verbosity.

Variables **`values`** (`list of tuples of values`) – The tuples of evaluation scores, one per metric, of each iteration. Empty list before this callback is called.

3.3 TimeCallback

`class a2pm.callbacks.TimeCallback`(*verbose=0*)

Bases: `a2pm.callbacks.base_callback.BaseCallback`

Time Attack Callback.

Records the time consumption of each iteration.

It is measured as nanoseconds per created example, according to the total samples that could be misclassified at each iteration.

Parameters **verbose** (*int, in {0, 1, 2} (default 0)*) – Verbosity level of the callback.

Set to 2 to enable a complete printing of the values and their descriptions, to 1 to enable a simple printing of the values, or to 0 to disable verbosity.

Variables **`values`** (`list of values`) – The time consumption of each iteration. Empty list before this callback is called.

CHAPTER

FOUR

PATTERNS

4.1 BasePattern

class a2pm.patterns.**BasePattern**(*features=None*, *probability=0.5*, *momentum=0.99*, *seed=None*)

Bases: `sklearn.base.BaseEstimator`

Base Perturbation Pattern.

A pattern analyzes specific feature subsets to fully or partially adapt itself, and then create valid and coherent perturbations in new data. This base class cannot be directly utilized.

It must be a class implementing the *fit*, *partial_fit* and *transform* methods, according to the following signatures:

fit(self, X, y=None) -> self

partial_fit(self, X, y=None) -> self

transform(self, X) -> numpy array

Parameters

- **features** (*int, array-like or None*) – Index or array-like of indices of features whose values are to be analyzed and perturbed.

  Set to None to use all features.

- **probability** (*float, in the (0.0, 1.0] interval*) – Probability of applying the pattern in *transform*.

  Set to 1 to always apply the pattern.

- **momentum** (*float, in the [0.0, 1.0] interval*) – Momentum of the *partial_fit* updates.

  Set to 1 to remain fully adapted to the initial data, without updates.

  Set to 0 to always fully adapt to new data, as in *fit*.

- **seed** (*int, None or a generator*) – Seed for reproducible random number generation.

  Set to None to disable reproducibility, or to a generator to use it unaltered.

Variables **generator** (*numpy generator object*) – The random number generator to be used by an inheriting class.

to_apply() → bool

Checks if the pattern is to be applied, according to the probability.

Returns True if the pattern is to be applied; False otherwise.

Return type bool

`fit_transform`(*X*, *y=None*) → numpy.ndarray

Fully adapts the pattern to new data, and then applies it to create data perturbations.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Input data.
- **y** (*ignored*) – Parameter compatibility.

Returns **X_perturbed** – Perturbed data.

Return type numpy array of shape (n_samples, n_features)

`partial_fit_transform`(*X*, *y=None*) → numpy.ndarray

Partially adapts the pattern to new data, according to the momentum, and then applies it to create data perturbations.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Input data.
- **y** (*ignored*) – Parameter compatibility.

Returns **X_perturbed** – Perturbed data.

Return type numpy array of shape (n_samples, n_features)

`set_params`(***params*)

Sets the parameters.

Parameters ****params** (*dict of 'parameter name - value' pairs*) – New valid parameters for this pattern.

Returns This pattern instance.

Return type self

`set_momentum`(*momentum*) → None

Sets the momentum.

Parameters **momentum** (*float, in the [0.0, 1.0] interval*) – Momentum of the *partial_fit* updates.

Set to 1 to remain fully adapted to the initial data, without updates.

Set to 0 to always fully adapt to new data, as in *fit*.

Raises `ValueError` – If the parameters do not fulfill the constraints.

`set_probability`(*probability*) → None

Sets the probability.

Parameters **probability** (*float, in the (0.0, 1.0] interval*) – Probability of applying the pattern in *transform*.

Set to 1 to always apply the pattern.

Raises `ValueError` – If the parameters do not fulfill the constraints.

`set_features`(*features*) → None

Sets the features.

Parameters **features** (*int, array-like or None*) – Index or array-like of indices of features whose values are to be perturbed.

Set to None to use all features.

Raises `ValueError` – If the parameters do not fulfill the constraints.

`set_seed`(*seed*) → None

Sets the seed for random number generation.

Parameters **seed** (*int, None or a generator*) – Seed for reproducible random number generation.

Set to None to disable reproducibility, or set to a generator to use it unaltered.

Raises `ValueError` – If the parameters do not fulfill the constraints.

4.2 CombinationPattern

`class a2pm.patterns.CombinationPattern`(*features=None*, *locked_features=None*, *probability=0.5*, *momentum=0.99*, *seed=None*)

Bases: `a2pm.patterns.base_pattern.BasePattern`

Combination Perturbation Pattern.

Perturbs features by replacing their values with other valid combinations. Intended use: categorical features (nominal and ordinal).

The valid combinations start being partially updated when the *partial_fit* or *partial_fit_transform* methods are called.

Parameters

- **features** (*int, array-like or None*) – Index or array-like of indices of features whose values are to be used in valid combinations.

  Set to None to use all features.

- **locked_features** (*int, array-like or None*) – Index or array-like of indices of features whose values are to be used in valid combinations, without being modified.

  These locked feature indices must also be present in the general *features* parameter.

  Set to None to not lock any feature.

- **probability** (*float, in the (0.0, 1.0] interval*) – Probability of applying the pattern in *transform*.

  Set to 1 to always apply the pattern.

- **momentum** (*float, in the [0.0, 1.0] interval*) – Momentum of the *partial_fit* updates.

  Set to 1 to remain fully adapted to the initial data, without updates.

  Set to 0 to always fully adapt to new data, as in *fit*.

- **seed** (*int, None or a generator*) – Seed for reproducible random number generation.

  Set to None to disable reproducibility, or to a generator to use it unaltered.

Variables

- **valid_cmbs** (*numpy array of combinations*) – The valid combinations recorded by the feature analysis of this pattern. Only available after a call to *fit* or *partial_fit*.
- **generator** (*numpy generator object*) – The random number generator used by this pattern.

`fit`(*X*, *y=None*)

Fully adapts the pattern to new data.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Input data.
- **y** (*ignored*) – Parameter compatibility.

Returns This pattern instance.

Return type self

`partial_fit`(*X*, *y=None*)

Partially adapts the pattern to new data.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Input data.
- **y** (*ignored*) – Parameter compatibility.

Returns This pattern instance.

Return type self

`transform`(*X*) → numpy.ndarray

Applies the pattern to create data perturbations.

Parameters **X** (*array-like of shape (n_samples, n_features)*) – Input data.

Returns **X_perturbed** – Perturbed data.

Return type numpy array of shape (n_samples, n_features)

`set_params`(***params*)

Sets the parameters.

Parameters ****params** (*dict of 'parameter name - value' pairs*) – Valid parameters for this pattern.

Returns This pattern instance.

Return type self

`set_locked_features`(*locked_features*) → None

Sets the locked features.

Parameters **locked_features** (*int, array-like or None*) – Index or array-like of indices of features whose values are to be used in valid combinations, without being modified.

These locked feature indices must also be present in the general *features* parameter.

Set to None to not lock any feature.

Raises `ValueError` – If the parameters do not fulfill the constraints.

4.3 IntervalPattern

`class a2pm.patterns.IntervalPattern`(*features=None*, *integer_features=None*, *ratio=0.1*, *max_ratio=None*, *missing_value=None*, *probability=0.5*, *momentum=0.99*, *seed=None*)

Bases: *`a2pm.patterns.base_pattern.BasePattern`*

Interval Perturbation Pattern.

Perturbs features by increasing or decreasing their values, according to a ratio of the valid interval of minimum and maximum values. Intended use: numerical features (continuous and discrete).

The valid interval starts being partially updated when the *partial_fit* or *partial_fit_transform* methods are called.

Parameters

- **features** (*int, array-like or None*) – Index or array-like of indices of features whose values are to be increased or decreased.

  Set to None to use all features.

- **integer_features** (*int, array-like or None*) – Index or array-like of indices of features whose values are to be increased or decreased, without a fractional part.

  These integer feature indices must also be present in the general *features* parameter.

  Set to None to not impose integer values on any feature.

- **ratio** (*float, > 0.0*) – Ratio of increase/decrease of the value of a feature, relative to its minimum and maximum values.

- **max_ratio** (*float or None, >= min_ratio*) – Maximum ratio. If provided, a random value in the *[ratio, max_ratio)* interval will be used.

  Set to None to always use the exact value of *ratio*.

- **missing_value** (*float or None*) – Value to be considered as missing when found in a feature, preventing its perturbation.

  Set to None to perturb all found values.

- **probability** (*float, in the (0.0, 1.0] interval*) – Probability of applying the pattern in *transform*.

  Set to 1 to always apply the pattern.

- **momentum** (*float, in the [0.0, 1.0] interval*) – Momentum of the *partial_fit* updates.

  Set to 1 to remain fully adapted to the initial data, without updates.

  Set to 0 to always fully adapt to new data, as in *fit*.

- **seed** (*int, None or a generator*) – Seed for reproducible random number generation.

  Set to None to disable reproducibility, or to a generator to use it unaltered.

Variables

- **`moving_mins`** (*`numpy array of numbers`*) – The minimum values recorded by the feature analysis of this pattern. Only available after a call to *fit* or *partial_fit*.

- **`moving_maxs`** (*`numpy array of numbers`*) – The maximum values recorded by the feature analysis of this pattern. Only available after a call to *fit* or *partial_fit*.

- `generator` (*numpy generator object*) – The random number generator used by this pattern.

`fit`(*X*, *y=None*)

Fully adapts the pattern to new data.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Input data.
- **y** (*ignored*) – Parameter compatibility.

Returns This pattern instance.

Return type self

`partial_fit`(*X*, *y=None*)

Partially adapts the pattern to new data.

Parameters

- **X** (*array-like of shape (n_samples, n_features)*) – Input data.
- **y** (*ignored*) – Parameter compatibility.

Returns This pattern instance.

Return type self

`transform`(*X*) → numpy.ndarray

Applies the pattern to create data perturbations.

Parameters **X** (*array-like of shape (n_samples, n_features)*) – Input data.

Returns **X_perturbed** – Perturbed data.

Return type numpy array of shape (n_samples, n_features)

`set_params`(***params*)

Sets the parameters.

Parameters ****params** (*dict of 'parameter name - value' pairs*) – Valid parameters for this pattern.

Returns This pattern instance.

Return type self

`set_missing_value`(*missing_value*) → None

Sets the missing value.

Parameters **missing_value** (*float or None*) – Value to be considered as missing when found in a feature, preventing its perturbation.

Set to None to perturb all found values.

Raises `ValueError` – If the parameters do not fulfill the constraints.

`set_ratio`(*ratio*, *max_ratio*) → None

Sets the ratio.

Parameters

- **ratio** (*float, > 0.0*) – Ratio of increase/decrease of the value of a feature, relative to its minimum and maximum values.

- **max_ratio** (*float or None, >= min_ratio*) – Maximum ratio. If provided, a random value in the *[ratio, max_ratio)* interval will be used.

  Set to None to always use the exact value of *ratio*.

Raises `ValueError` – If the parameters do not fulfill the constraints.

`set_integer_features`(*integer_features*) → None

Sets the integer features.

Parameters **integer_features** (*int, array-like or None*) – Index or array-like of indices of features whose values are to be increased or decreased, without a fractional part.

These integer feature indices must also be present in the general *features* parameter.

Set to None to not impose integer values on any feature.

Raises `ValueError` – If the parameters do not fulfill the constraints.

CHAPTER

FIVE

WRAPPERS

5.1 BaseWrapper

`class a2pm.wrappers.`**`BaseWrapper`**`(`*`**params`*`)`

Bases: `object`

Base Classifier Wrapper.

A wrapper encapsulates a classifier that is ready to provide class predictions (is already fitted) for the *generate* method. This base class cannot be directly utilized.

Additionally, a wrapped classifier can also be used as a *class_discriminator* function, to be called to identify the Class ID of each sample.

It must be a class implementing the *predict* method, according to the following signature:

predict(self, X) -> y

Parameters ****params** (*dict of 'parameter name - value' pairs*) – Optional parameters to provide to the classifier during the class prediction process.

5.2 KerasWrapper

`class a2pm.wrappers.`**`KerasWrapper`**`(`*`classifier`*`, `*`classes=None`*`, `*`**params`*`)`

Bases: *`a2pm.wrappers.base_wrapper.BaseWrapper`*

Keras Classifier Wrapper.

Encapsulates a Tensorflow/Keras classification model.

Parameters

- **classifier** (object with a *predict* method) – Fitted classifier to be wrapped.
- **classes** (*list of Class IDs or None (default None)*) – Classes to convert predictions to, using the indices provided by the prediction process.

  Set to None to use the default class indices.
- ****params** (*dict of 'parameter name - value' pairs*) – Optional parameters to provide to the classifier during the prediction process.

`predict``(`*`X`*`)`

Applies the wrapped classifier and converts its class probability predictions.

Parameters **X** (*array-like in the (n_samples, n_features) shape*) – Input data.

Returns **y** – The class predictions.

Return type numpy array of shape (n_samples,)

5.3 SklearnWrapper

`class a2pm.wrappers.SklearnWrapper(classifier, **params)`

Bases: `a2pm.wrappers.base_wrapper.BaseWrapper`

Sklearn Classifier Wrapper.

Encapsulates a Scikit-Learn classification model.

Parameters

- **classifier** (object with a *predict* method) – Fitted classifier to be wrapped.
- ****params** (*dict of 'parameter name - value' pairs*) – Optional parameters to provide to the classifier during the prediction process.

`predict(X)`

Applies the wrapped classifier directly, without needing to convert its class predictions.

Parameters **X** (*array-like in the (n_samples, n_features) shape*) – Input data.

Returns **y** – The class predictions.

Return type numpy array of shape (n_samples,)

5.4 TorchWrapper

`class a2pm.wrappers.TorchWrapper(classifier, classes=None, **params)`

Bases: `a2pm.wrappers.base_wrapper.BaseWrapper`

Torch Classifier Wrapper.

Encapsulates a PyTorch classification model.

Parameters

- **classifier** (object with a *__call__* method) – Fitted classifier to be wrapped.
- **classes** (*list of Class IDs or None (default None)*) – Classes to convert predictions to, using the indices provided by the prediction process.

  Set to None to use the default class indices.
- ****params** (*dict of 'parameter name - value' pairs*) – Optional parameters to provide to the classifier during the prediction process.

`predict(X)`

Applies the wrapped classifier and converts its class probability predictions.

Parameters **X** (*array-like in the (n_samples, n_features) shape*) – Input data.

Returns **y** – The class predictions.

Return type numpy array of shape (n_samples,)

CHAPTER

SIX

INDEX

INDEX