# mloptimizer

***Release 0.8.5***

**Antonio Caparrini, Javier Arroyo**

**Apr 06, 2024**

# HOME:

CHAPTER

ONE

# MLOPTIMIZER


**mloptimizer** is a Python library for optimizing hyperparameters of machine learning algorithms using genetic algorithms. With mloptimizer, you can find the optimal set of hyperparameters for a given machine learning model and dataset, which can significantly improve the performance of the model. The library supports several popular machine learning algorithms, including decision trees, random forests, and gradient boosting classifiers. The genetic algorithm used in mloptimizer provides an efficient and flexible approach to search for the optimal hyperparameters in a large search space.

## 1.1 Features

- Easy to use
- DEAP-based genetic algorithm ready to use with several machine learning algorithms
- Adaptable to use with any machine learning algorithm that complies with the Scikit-Learn API
- Default hyperparameter ranges
- Default score functions for evaluating the performance of the model
- Reproducibility of results

## 1.2 Advanced Features

- Extensible with more machine learning algorithms that comply with the Scikit-Learn API
- Customizable hyperparameter ranges
- Customizable score functions
- Optional mlflow compatibility for tracking the optimization process

## 1.3 Installation

It is recommended to create a virtual environment using the `venv` package. To learn more about how to use `venv`, check out the official Python documentation at https://docs.python.org/3/library/venv.html.

```
# Create the virtual environment
python -m venv myenv
# Activate the virtual environment
source myenv/bin/activate
```

To install `mloptimizer`, run:

```
pip install mloptimizer
```

You can get more information about the package installation at https://pypi.org/project/mloptimizer/.

### 1.3.1 Quickstart

Here's a simple example of how to optimize hyperparameters in a decision tree classifier using the iris dataset:

```
from mloptimizer.core import Optimizer
from mloptimizer.hyperparams import HyperparameterSpace
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_iris

# 1) Load the dataset and get the features and target
X, y = load_iris(return_X_y=True)

# 2) Define the hyperparameter space (a default space is provided for some algorithms)
hyperparameter_space = HyperparameterSpace.get_default_hyperparameter_
↪space(DecisionTreeClassifier)

# 3) Create the optimizer and optimize the classifier
opt = Optimizer(estimator_class=DecisionTreeClassifier, features=X, labels=y, hyperparam_
↪space=hyperparameter_space)

# 4) Optimize the classifier, the optimization returns the best estimator found in the␣
↪optimization process
# - 10 generations starting with a population of 10 individuals, other parameters are␣
↪set to default
clf = opt.optimize_clf(population_size=10, generations=10)
```

Other algorithms can be used, such as `RandomForestClassifier` or `XGBClassifier` which have a default hyperparameter space defined in the library. Even if the algorithm is not included in the default hyperparameter space, you can define your own hyperparameter space following the documentation.

The optimization will create a directory in the current folder with a name like `YYYYMMDD_nnnnnnnnnn_SklearnOptimizer`. This folder contains the results of the optimization, including the best estimator found and the log file `opt.log` informing with all the steps, the best estimator and the result of the optimization.

A structure like this will be created:

```
├── checkpoints
│   ├── cp_gen_0.pkl
│   └── cp_gen_1.pkl
├── graphics
│   ├── logbook.html
│   └── search_space.html
├── opt.log
├── progress
│   ├── Generation_0.csv
│   └── Generation_1.csv
└── results
    ├── logbook.csv
    └── populations.csv
```

Each item in the directory is described below:

- `checkpoints`: This directory contains the checkpoint files for each generation of the genetic optimization process. These files are used to save the state of the optimization process at each generation, allowing for the process to be resumed from a specific point if needed.
  - `cp_gen_0.pkl`, `cp_gen_1.pkl`: These are the individual checkpoint files for each generation. They are named according to the generation number and are saved in Python's pickle format.
- `graphics`: This directory contains HTML files for visualizing the optimization process.
  - `logbook.html`: This file provides a graphical representation of the logbook, which records the statistics of the optimization process over generations.
  - `search_space.html`: This file provides a graphical representation of the search space of the optimization process.
- `opt.log`: This is the log file for the optimization process. It contains detailed logs of the optimization process, including the performance of the algorithm at each generation.
- `progress`: This directory contains CSV files that record the progress of the optimization process for each generation.
  - `Generation_0.csv`, `Generation_1.csv`: These are the individual progress files for each generation. They contain detailed information about each individual in the population at each generation.
- `results`: This directory contains CSV files with the results of the optimization process.
  - `logbook.csv`: This file is a CSV representation of the logbook, which records the statistics of the optimization process over generations.
  - `populations.csv`: This file contains the final populations of the optimization process. It includes the hyperparameters and fitness values of each individual in the population.

More details in the documentation.

## 1.4 Examples

Examples can be found in examples on readthedocs.io.

## 1.5 Dependencies

The following dependencies are used in `mloptimizer`:

- Deap - Genetic Algorithms
- XGBoost - Gradient boosting classifier
- Scikit-Learn - Machine learning algorithms and utilities

Optional:

- Keras - Deep learning library
- mlflow - Tracking the optimization process

## 1.6 Documentation

The documentation for `mloptimizer` can be found in the project's wiki with examples, classes and methods reference.

## 1.7 Authors

- **Antonio Caparrini** - *Author* - caparrini
- **Javier Arroyo Gallardo** - *Author* - javiag

## 1.8 License

This project is licensed under the MIT License.

## 1.9 FAQs

- TODO

CHAPTER

TWO

# INTRODUCTION

This user guide is an introduction to the MLOptimizer library, designed to optimize machine learning models with a focus on ease of use of the Deap library. The guide will demonstrate the library's capabilities through examples and highlight its features and customization options.

MLOptimizer is intended to complement detailed API documentation, offering practical insights and optimal usage strategies.

While MLOptimizer integrates seamlessly with Python's machine learning ecosystem, it's built on Deap optimization algorithms, which are not specific to machine learning. This guide primarily uses Python examples, providing a straightforward path for practitioners familiar with Python-based machine learning libraries.

## 2.1 Features

The goal of mloptimizer is to provide a user-friendly, yet powerful optimization tool that:

- Easy to use
- DEAP-based genetic algorithm ready to use with several machine learning algorithms
- Compatible with any machine learning algorithm that complies with the Scikit-Learn API
- Default hyperparameter spaces for the most common machine learning algorithms
- Default score functions for evaluating the performance of the model
- Reproducibility of results
- Extensible with more machine learning algorithms that comply with the Scikit-Learn API
- Customizable hyperparameter ranges
- Customizable score functions

## 2.2 Using mloptimizer

### 2.2.1 Step 1: Select and Setup the Algorithm to Optimize

MLOptimizer uses a wrapper, *SklearnOptimizer*, for the algorithm for classification or regression that is going to be optimized. It currently have default hyperparameter spaces the following algorithms:

- *DecisionTreeClassifier*: Decision Tree Classifier from scikit-learn
- *RandomForestClassifier*: Random Forest Classifier from scikit-learn

- *ExtraTreesClassifier*: Extra Trees Classifier from scikit-learn
- *GradientBoostingClassifier*: Gradient Boosting Classifier from scikit-learn
- *SVC*: Support Vector Classifier from scikit-learn
- *KerasClassifier*: Custom Keras Classifier class
- *XGBClassifier*: XGBoost Classifier

Let's assume that we want to fine-tune the decision tree classifier from scikit-learn, wrapped in *SklearnOptimizer*.

To instantiate the wrapper, you need to specify the class of the machine learning algorithm, the dataset to work with, the hyperparameter space (fixed and evolvable), the input features (as a matrix), and the output features (as a column).

The wrapper has a variable with the set of hyperparameters to be explored. For the case of the decision tree classifier in *DecisionTreeClassifier* from *sklearn.tree* the default hyperparameters and their exploration ranges are:

- *min_samples_split*, range [2, 50]
- *min_samples_leaf*, range [1, 20]
- *max_depth*, range [2, 20]
- *min_impurity_decrease*, range [0, 0.15] in 1000 steps
- *ccp_alpha*, range [0, 0.003] in 100,000 steps

For a quick start, we will explore the default hyperparameters using a default range for exploring each of them.

Similarly, in the wrapper, you can set up the metric to be optimized (the parameter is called *score_function* and the default value is accuracy) from the metrics available in scikit-learn (*sklearn.metrics*) and the evaluation setting (the parameter is called *model_evaluation* and the default value is the *train_score*).

See the API reference for more details on setting up the wrapper and the optimization.

### 2.2.2 Step 2: Running the Optimization

Once you have instantiated the wrapper with the algorithm to optimize, you can run the genetic optimization.

Typically, you should set the number of generations (3 by default) and the size of the population (10 by default).

The optimization returns the best classifier found during the genetic optimization, tuned with the corresponding hyperparameters. Additionally, during the optimization, a structure of directories is created to store the results of the optimization process. The structure of the directories is explained in the section on the optimizer output directory structure and contain useful information, logs, checkpoints and plots.

### 2.2.3 Step 3: Using the Outcome of the Optimization Process

The result of the optimization process is the optimal classifier object. You can use this object to make predictions on your dataset. For example, if *clf_result* is the returned classifier, you can use *clf_result.predict(X)* to make predictions.

In addition to the optimal classifier, you can explore the outcomes of the optimization process, such as the evolution of the population and the best score at each generation. These outcomes are stored in the directory created by the optimizer, as explained in the section on the optimizer output directory structure.

**Warning:** mloptimizer is not a machine learning library. It is a hyperparameter optimization library that can be used with any machine learning library that complies with the scikit-learn API.

**Warning:** Before optimizing a machine learning model using mloptimizer it is recommended first to have a cleaned dataset. mloptimizer does not provide any data preprocessing or cleaning tools.

**Note:** The examples in this guide are aligned with the latest version of mloptimizer. Users are encouraged to ensure they are using the most recent release to fully leverage the library's capabilities.

CHAPTER

THREE

# INSTALLATION

This package is available in the Python Package Index. So the easiest way to install it is using `pip`:

```
pip install mloptimizer
```

**Warning:** It is recommended to install `mloptimizer` in a virtual environment. See the section below for more information.

It is also available at Github where you can clone it.

## 3.1 Virtual environment

It is recommended to create a virtual environment using the `venv` package. To learn more about how to use `venv`, check out the official Python documentation at https://docs.python.org/3/library/venv.html. Once installed, you can create a virtual environment and activate it using the following commands:

```
# Create the virtual environment
python -m venv myenv
# Activate the virtual environment
source myenv/bin/activate
```

To install `mloptimizer`, then simply run:

```
pip install mloptimizer
```

CHAPTER

FOUR

# BASICS

The BaseOptimizer is an abstract base class that provides the fundamental structure for all optimizers in the mloptimizer package. It is designed to optimize a classifier using a genetic algorithm. The class includes methods for setting up the optimization process, defining the hyperparameters to be optimized, and running the optimization.

The BaseOptimizer class is designed to be subclassed. MLOptimizer provides several subclasses of the BaseOptimizer class.

## 4.1 Overview

### 4.1.1 Introduction

The main class objects are the *Optimizer* and the *HyperparameterSpace* classes.

The optimizer *Optimizer* is able to optimize any model that complies with the *sklearn* API. The *HyperparameterSpace* class is used to define the hyperparameters that will be optimized, either the fixed hyperparameters or the hyperparameters that will be optimized.

### 4.1.2 Usage

To use the *Optimizer* class:

1. Define your features and labels.

2. Choose a model to optimize that complies with the *sklearn* API. (e.g. *XGBClassifier*). 2. Create an instance of *HyperparameterSpace* with the hyperparameters that you want to optimize. 3. Call the *optimize_clf()* method to start the optimization process.

---

**Note:** There are default HyperparameterSpaces defined in the `conf` folder for the most common models. You can use the HyperparameterSpace.get_default_hyperparams(class) (class e.g. XGBClassifier).

---

There are several parameters than can be passed to the *Optimizer* constructor:

- *estimator_class*: The class of the model to optimize. It should comply with the *sklearn* API.
- *X*: The features of your dataset.
- *y*: The labels of your dataset.
- *folder*: The folder where the files and folder will be saved. Defaults to the current directory.
- *log_file*: The name of the log file. Defaults to *mloptimizer.log*.

- *hyperparam_space*: The hyperparameter space to use for the optimization process.
- *eval_function*: The function to use to evaluate the model. Defaults to *train_score*.
- *score_function*: The function to use to score the model. Defaults to *accuracy_score*.
- *seed*: The seed to use for reproducibility. Defaults to a random integer between 0 and 1000000.

### 4.1.3 Default Usage Example

The simplest example of using the Optimizer is:

- Store your features and labels in *X* and *y* respectively.
- Use HyperparameterSpace.get_default_hyperparams(XGBClassifier) to get the default hyperparameters for the model you want to optimize.
- Create an instance of *Optimizer* with your classifier class, hyperparameter space, data and leave all other parameters to their default values.
- Call the *optimize_clf()* method to start the optimization process. You can pass the population size and the number of generations to the method.
- The result of the optimization process will be a object of type XGBClassifier with the best hyperparameters found.

```
from mloptimizer.core import Optimizer
from mloptimizer.hyperparams import HyperparameterSpace
from xgboost import XGBClassifier
from sklearn.datasets import load_iris

# 1) Load the dataset and get the features and target
X, y = load_iris(return_X_y=True)

# 2) Define the hyperparameter space (a default space is provided for some algorithms)
hyperparameter_space = HyperparameterSpace.get_default_hyperparameter_
↪space(XGBClassifier)

# 3) Create the optimizer and optimize the classifier
opt = Optimizer(estimator_class=XGBClassifier, features=X, labels=y, hyperparam_
↪space=hyperparameter_space)

clf = opt.optimize_clf(10, 10)
```

This will create a folder (in the current location) with name *YYYYMMDD_nnnnnnnnnn_Optimizer* (where *YYYYMMDD_nnnnnnnnnn* is the current timestamp) and a log file named *mloptimizer.log*. To inspect the structure of the folder and what can you find in it, please refer to the *Folder Structure* section.

### 4.1.4 Custom HyperparameterSpace Example

Among the parameters that can be passed to the *Optimizer* constructor, the *hyperaram_space* of class *HyperparameterSpace* is really important and should be aligned with the machine learning algorithm passed to the Optimizer: *fixed_hyperparams* and *evolvable_hyperparams*.

The *evolvable_hyperparams* parameter is a dictionary of custom hyperparameters. The key of each hyperparameter is the name of the hyperparameter, and the value is the *Hyperparam* object itself. To understand how to use the *Hyperparam* object, please refer to the *Hyperparam* section inside Concepts.

The *fixed_hyperparams* parameter is a dictionary of fixed hyperparameters. This is simply a dictionary where the key is the name of the hyperparameter, and the value is the value of the hyperparameter. These hyperparameters will not be optimized, but will be used as fixed values during the optimization process.

An example of using custom hyperparameters is:

```
from mloptimizer.hyperparams import Hyperparam, HyperparameterSpace
# Define your custom hyperparameters
fixed_hyperparams = {
    'max_depth': 5
}
evolvable_hyperparams = {
    'colsample_bytree': Hyperparam("colsample_bytree", 3, 10, 'float', 10),
    'gamma': Hyperparam("gamma", 0, 20, 'int'),
    'learning_rate': Hyperparam("learning_rate", 1, 100, 'float', 1000),
    # 'max_depth': Hyperparam("max_depth", 3, 20, 'int'),
    'n_estimators': Hyperparam("n_estimators", 100, 500, 'int'),
    'subsample': Hyperparam("subsample", 700, 1000, 'float', 1000),
    'scale_pos_weight': Hyperparam("scale_pos_weight", 15, 40, 'float', 100)
}


custom_hyperparam_space = HyperparameterSpace(fixed_hyperparams, evolvable_hyperparams)

# Create an instance of XGBClassifierOptimizer with custom hyperparameters
xgb_optimizer = Optimizer(estimator_class=XGBClassifier,features=X, labels=y,
                          hyperparam_space=custom_hyperparam_space)

# Start the optimization process
result = xgb_optimizer.optimize_clf(3, 3)
```

Both *evolvable_hyperparams* and *fixed_hyperparams* can be used together, providing several different ways to customize the optimization process.

### 4.1.5 Reproducibility

Researchers often need to be able to reproduce their results. During the research process it could be advisable to run several optimizations processes with different parameters or input data. However, if the results of the optimization process are not reproducible, it will be difficult to compare the results of the different optimization processes. In order to make the results reproducible, the *Optimizer* have a *seed* parameter. This parameter is used to set the seed of the random number generator used during the optimization process. If you set the same seed, the results of the optimization process will be the same.

An example of two executions of the optimization process with the same seed that will produce the same result is:

```python
from mloptimizer.core import Optimizer
from mloptimizer.hyperparams import HyperparameterSpace
from xgboost import XGBClassifier
from sklearn.datasets import load_iris

# 1) Load the dataset and get the features and target
X, y = load_iris(return_X_y=True)

# 2) Define the hyperparameter space (a default space is provided for some algorithms)
hyperparameter_space = HyperparameterSpace.get_default_hyperparameter_
↪space(XGBClassifier)

# 3) Create two instances of Optimizer with the same seed
xgb_optimizer1 = Optimizer(estimator_class=XGBClassifier, features=X, labels=y,
                           hyperparam_space = hyperparameter_space, seed=42)
result1 = xgb_optimizer1.optimize_clf(3, 3)

xgb_optimizer2 = Optimizer(estimator_class=XGBClassifier, features=X, labels=y,
                           hyperparam_space = hyperparameter_space, seed=42)
result2 = xgb_optimizer2.optimize_clf(3, 3)

# Verify that the results are the same
# The comparison is done using the string representation of the result objects
# which are the hyperparameters of the best model found
assert str(result1)== str(result2)
```

## 4.2 Optimizer Directory Structure

When an optimizer is run, it generates a directory in the current working directory (or the given directory as input). This directory, named in the format *YYYYMMDD_nnnnnnnnnn_OptimizerName*, contains the results of the optimization process, including the best estimator found, a log file detailing the optimization steps, and the final result of the optimization.

### 4.2.1 Directory Structure

The directory structure is as follows:

```
├── checkpoints
│   ├── cp_gen_0.pkl
│   └── cp_gen_1.pkl
├── graphics
│   ├── logbook.html
│   └── search_space.html
├── opt.log
├── progress
│   ├── Generation_0.csv
│   └── Generation_1.csv
└── results
    ├── logbook.csv
    └── populations.csv
```

### 4.2.2 Directory Contents

Each item in the directory serves a specific purpose:

- ***checkpoints*: Contains the checkpoint files for each generation of the genetic optimization process. These files preserve the state of the optimization process at each generation, enabling the process to be resumed from a specific point if necessary.**
  - *cp_gen_0.pkl*, *cp_gen_1.pkl*: These are the individual checkpoint files for each generation. They are named according to the generation number and are saved in Python's pickle format.
- ***graphics*: Contains HTML files for visualizing the optimization process.**
  - *logbook.html*: Provides a graphical representation of the logbook, which records the statistics of the optimization process over generations.
  - *search_space.html*: Provides a graphical representation of the search space of the optimization process.
- *opt.log*: The log file for the optimization process. It contains detailed logs of the optimization process, including the performance of the algorithm at each generation.
- ***progress*: Contains CSV files that record the progress of the optimization process for each generation.**
  - *Generation_0.csv*, *Generation_1.csv*: These are the individual progress files for each generation. They contain detailed information about each individual in the population at each generation.
- ***results*: Contains CSV files with the results of the optimization process.**
  - *logbook.csv*: This file is a CSV representation of the logbook, which records the statistics of the optimization process over generations.
  - *populations.csv*: This file contains the final populations of the optimization process. It includes the hyperparameters and fitness values of each individual in the population.

CHAPTER

FIVE

# CONCEPTS

Concepts are the building blocks of the hyperparameter optimization framework. They are used to define the search space and the score function.

## 5.1 Hyperparam Class

The Hyperparam class is a crucial component of our library, designed to optimize the hyperparameters of machine learning algorithms using the DEAP library, which provides genetic algorithms.

### 5.1.1 Why We Need the Hyperparam Class

In the context of genetic optimization, a common problem is the repeated evaluation of slightly different individuals. This can lead to inefficiencies in the optimization process. To mitigate this, we use the Hyperparam class to limit the values of the hyperparameters, ensuring that the same individuals are not evaluated multiple times.

### 5.1.2 How It Is Used

The Hyperparam class is used to define a hyperparameter to optimize. It includes the name, minimum value, maximum value, and type of the hyperparameter. This class also controls the precision of the hyperparameter to avoid multiple evaluations with close values due to decimal positions.

The Hyperparam class has several methods, including:

- *__init__*: Initializes a new instance of the Hyperparam class.
- *correct*: Returns the real value of the hyperparameter in case some mutation could surpass the limits.
- *__eq__*: Overrides the default implementation to compare two Hyperparam instances.
- *__str__* and *__repr__*: Overrides the default implementations to provide a string representation of the Hyperparam instance.

### 5.1.3 Types of Hyperparam

The *Hyperparam* class supports several types of hyperparameters. Here are examples of each type:

- Integer hyperparameter:

```
hyperparam_int = Hyperparam(name='max_depth', min_value=1,
                            max_value=10, hyperparam_type='int')
```

- Float hyperparameter:

```
hyperparam_float = Hyperparam(name='learning_rate', min_value=0.01, max_value=1.0,
                              hyperparam_type='float', scale=100)
```

- 'nexp' hyperparameter:

```
hyperparam_nexp = Hyperparam(name='nexp_param', min_value=1,
                             max_value=100, hyperparam_type='nexp')
```

- 'x10' hyperparameter:

```
hyperparam_x10 = Hyperparam(name='x10_param', min_value=1,
                            max_value=100, hyperparam_type='x10')
```

In these examples, we define hyperparameters of different types. The 'nexp' and 'x10' types are special types that apply a transformation to the hyperparameter value.

### 5.1.4 Examples

Here's an example of how to use the Hyperparam class:

```
# Define a hyperparameter
hyperparam = Hyperparam(name='learning_rate', min_value=0, max_value=1,
                        hyperparam_type='float', scale=100)

# Correct a value
# This will return 1.0 as 150 is beyond the max_value
corrected_value = hyperparam.correct(150)
```

In this example, we define a hyperparameter named 'learning_rate' with a minimum value of 0, a maximum value of 1, and a type of float. The 'correct' method is then used to correct a value that is beyond the defined maximum value.

Here's an example of how you can create a *HyperparameterSpace* instance and pass custom hyperparameters to it:

```
from mloptimizer.hyperparams import Hyperparam, HyperparameterSpace

# Define custom hyperparameters
fixed_hyperparams = {
    "criterion": "gini"
}
evolvable_hyperparams = {
    "min_samples_split": Hyperparam("min_samples_split", 2, 50, 'int'),
    "min_samples_leaf": Hyperparam("min_samples_leaf", 1, 20, 'int'),
    "max_depth": Hyperparam("max_depth", 2, 20, 'int'),
```

(continues on next page)

(continued from previous page)

```
    "min_impurity_decrease": Hyperparam("min_impurity_decrease", 0, 150, 'float', 1000),
    "ccp_alpha": Hyperparam("ccp_alpha", 0, 300, 'float', 100000)
}


# Create a HyperparameterSpace instance
hyperparam_space = HyperparameterSpace(fixed_hyperparams, evolvable_hyperparams)

# Then we can use the hyperparam_space instance to optimize the hyperparameters
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_iris
from mloptimizer.core import Optimizer

# Load the iris dataset
X,y = load_iris(return_X_y=True)

tree_optimizer = Optimizer(estimator_class=DecisionTreeClassifier,
                           hyperparam_space=hyperparam_space,
                           features=X, labels=y)
tree_optimizer.optimize_clf(3, 3)
```

In this example, we define custom hyperparameters and create a *HyperparameterSpace* instance. We then use the *HyperparameterSpace* instance to optimize the hyperparameters of a *DecisionTreeClassifier* using the *Optimizer* class.

## 5.2 Score Functions (NEED UPDATE)

The *model_evaluation.py* module in our library provides several score functions that are used to evaluate the performance of machine learning algorithms. These score functions are crucial in the context of genetic optimization, where they serve as fitness values. In genetic optimization, a fitness value determines how well an individual (in this case, a machine learning algorithm defined by its hyperparameters) performs in a given generation. The better the fitness value, the more likely the individual is to survive and reproduce in the next generation.

A score function takes as input:

- The true labels of the data
- The predicted labels of the data
- A machine learning algorithm complying with the scikit-learn API
- A scoring function metric (e.g. accuracy, precision, recall, F1 score, etc.)

---

**Note:** The library provides several score functions that can be used for genetic optimization. However, users can also define their own score functions if they wish to do so.

---

### 5.2.1 Score Functions

The *model_evaluation.py* module provides the following score functions:

- *train_score*: This function trains a classifier with the provided features and labels, and then calculates a score using the provided score function.
- *train_test_score*: This function splits the provided features and labels into a training set and a test set. It then trains a classifier on the training set and calculates a score on the test set using the provided score function.
- *kfold_score*: This function evaluates a classifier using K-Fold cross-validation. It splits the provided features and labels into K folds, trains a classifier on K-1 folds, and calculates a score on the remaining fold. This process is repeated K times, and the function returns the average score across all folds.
- *kfold_stratified_score*: This function is similar to *kfold_score*, but it uses stratified K-Fold cross-validation. This means that it preserves the percentage of samples for each class in each fold.
- *temporal_kfold_score*: This function is similar to *kfold_score*, but it uses temporal K-Fold cross-validation. This means that it respects the order of the data, making it suitable for time series data.

Each of these score functions takes a classifier, features, and labels as input. They also take a score function as input, which is used to calculate the score. The score function could be any function that takes the true labels and the predicted labels as input and returns a score. Examples of score functions include accuracy, precision, recall, F1 score, etc.

### 5.2.2 Examples

Here's an example of how to use the *train_score* function:

```
from mloptimizer.evaluation import model_evaluation
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Define features, labels, and classifier
from sklearn.datasets import load_iris
features, labels = load_iris(return_X_y=True)
clf = RandomForestClassifier()

# Use the train_score function
score = model_evaluation.train_score(features, labels, clf, metrics={"accuracy":␣
↪accuracy_score})
```

In this example, we first define the features, labels, and classifier. We then use the *train_score* function to train the classifier and calculate the score. The *accuracy_score* function from *sklearn.metrics* is used as the score function.

## 5.3 Reproducibility

Reproducibility is a key aspect of scientific research, and more precisely, in machine learning. `MLOptimizer` provides an input parameter `seed` that allows to set the random seed for:

- The random number generator of the optimizer generating the initial population and the mutations
- The random number generator of the model on training
- The random number generator of the data on split

An example of usage is:

```
from sklearn.datasets import load_breast_cancer as dataset
from sklearn.tree import DecisionTreeClassifier
from mloptimizer.core import Optimizer
from mloptimizer.hyperparams import HyperparameterSpace

X, y = load_iris(return_X_y=True)
default_hyperparam_space = HyperparameterSpace.get_default_hyperparameter_
↪space(DecisionTreeClassifier)
population = 2
generations = 2
seed = 25
distinct_seed = 2
# It is important to run the optimization
# right after the creation of the optimizer
optimizer1 = Optimizer(estimator_class=DecisionTreeClassifier, features=X, labels=y,
                       hyperparam_space=default_hyperparam_space, seed=seed)
result1 = optimizer1.optimize_clf(population_size=population,
                                  generations=generations)
# WARNING: In case the optimizer2 would be created after the optimizer1,
# the results would be different
optimizer2 = Optimizer(estimator_class=DecisionTreeClassifier, features=X, labels=y,
                       hyperparam_space=default_hyperparam_space, seed=seed)
result2 = optimizer2.optimize_clf(population_size=population,
                                  generations=generations)

optimizer3 = Optimizer(estimator_class=DecisionTreeClassifier, features=X, labels=y,
                       hyperparam_space=default_hyperparam_space, seed=distinct_seed)
result3 = optimizer3.optimize_clf(population_size=population,
                                  generations=generations)
str(result1) == str(result2)
str(result1) != str(result3)
```

**Warning:** To ensure reproducibility, it is important to run the optimization right after the creation of the optimizer with the seed to ensure no other random number generator has been used in the meantime.

## 5.4 Parallel processing

Relying on the Deap capability to parallelize the evaluation of the fitness function, we can use the `multiprocessing` module to parallelize the evaluation of the fitness function. This is done passing the `use_parallel` parameter as `True` to initialize the `Optimizer` object. This parameter is set to `False` by default.

An example of the speedup that can be achieved using parallel processing is shown below.

**Note:** In the example below, the seed is set to 25 to ensure the result using parallel processing is the same as the one without parallel processing.

**Warning:** Parallel processing is not supported for the `XGBClassifier` and `KerasClassifier` classifiers.

```
from mloptimizer.core import Optimizer
from mloptimizer.hyperparams import HyperparameterSpace
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_iris
import time

# Load the dataset and get the features and target
X, y = load_iris(return_X_y=True)

# Define the hyperparameter space (a default space is provided for some algorithms)
hyperparameter_space = HyperparameterSpace.get_default_hyperparameter_
↪space(DecisionTreeClassifier)

# Set the seed to ensure the result using parallel processing is the same as the one␣
↪without parallel processing
my_seed = 25
population = 50
generations = 4

opt_with_parallel = Optimizer(estimator_class=DecisionTreeClassifier, features=X,␣
↪labels=y,
                              hyperparam_space=hyperparameter_space, seed=my_seed, use_
↪parallel=True)

start_time_parallel = time.time()
clf_with_parallel = opt_with_parallel.optimize_clf(population, generations)
end_time_parallel = time.time()

opt = Optimizer(estimator_class=DecisionTreeClassifier, features=X, labels=y,
                hyperparam_space=hyperparameter_space, seed=my_seed, use_parallel=False)
start_time = time.time()
clf = opt.optimize_clf(population, generations)
end_time = time.time()

elapsed_time_parallel = end_time_parallel - start_time_parallel
elapsed_time = end_time - start_time
speedup = round(((elapsed_time_parallel / elapsed_time) - 1) * 100, 2)

print(f"Elapsed time with parallel: {elapsed_time_parallel}")
print(f"Elapsed time without parallel: {elapsed_time}")
print(f"Speedup: {speedup}%")
```

CHAPTER

SIX

# EXAMPLES

Examples of how to use the mloptimizer library.

## 6.1 Evolution (logbook) graph

mloptimizer provides a function to plot the evolution of the fitness function.

```
from mloptimizer.core import Optimizer
from mloptimizer.hyperparams import HyperparameterSpace
from sklearn.tree import DecisionTreeClassifier
from mloptimizer.aux.plots import plotly_logbook
import plotly
import os
from sklearn.datasets import load_iris
```

Load the iris dataset to obtain a vector of features X and a vector of labels y. Another dataset or a custom one can be used

```
X, y = load_iris(return_X_y=True)
```

Define the HyperparameterSpace, you can use the default hyperparameters for the machine learning model that you want to optimize. In this case we use the default hyperparameters for a DecisionTreeClassifier. Another dataset or a custom one can be used

```
hyperparam_space = HyperparameterSpace.get_default_hyperparameter_
↪space(DecisionTreeClassifier)
```

We use the default TreeOptimizer class to optimize a decision tree classifier.

```
opt = Optimizer(estimator_class=DecisionTreeClassifier, features=X, labels=y,
                hyperparam_space=hyperparam_space, folder="Evolution_example")
```

```
ERROR:root:The folder Evolution_example could not be created.
```

To optimizer the classifier we need to call the optimize_clf method. The first argument is the number of generations and the second is the number of individuals in each generation.

```
clf = opt.optimize_clf(10, 10)
```

```
INFO:mloptimizer.log:Initiating genetic optimization...
INFO:mloptimizer.log:Algorithm: Optimizer

Genetic execution:   0%|          | 0/11 [00:00<?, ?it/s]
Genetic execution:  55%|    | 6/11 [00:00<00:00, 51.16it/s]
Genetic execution: 100%|| 11/11 [00:00<00:00, 45.31it/s]
```

We can plot the evolution of the fitness function. The black lines represent the max and min fitness values across all generations. The green, red and blue line are respectively the max, min and avg fitness value for each generation. Each grey point in the graph represents an individual.

```
population_df = opt.runs[-1].population_2_df()
g_logbook = plotly_logbook(opt.logbook, population_df)
plotly.io.show(g_logbook)
```

At the end of the evolution the graph is saved as an html at the path:

```
print(opt.tracker.graphics_path)
print(os.listdir(opt.tracker.graphics_path))
```

```
Evolution_example/20240406_161630_Tracker/graphics
['logbook.html', 'search_space.html']
```

The data to generate the graph is available at the path:

```
print(opt.tracker.results_path)
print(os.listdir(opt.tracker.results_path))

del opt
```

```
Evolution_example/20240406_161630_Tracker/results
['populations.csv', 'logbook.csv']
```

**Total running time of the script:** (0 minutes 3.966 seconds)

## 6.2 Search space graph

mloptimizer provides a function to plot the search space of the optimization.

```
from mloptimizer.core import Optimizer
from mloptimizer.hyperparams import HyperparameterSpace
from sklearn.tree import DecisionTreeClassifier
from mloptimizer.aux.plots import plotly_search_space
import plotly
import os
from sklearn.datasets import load_iris
```

Load the iris dataset to obtain a vector of features X and a vector of labels y. Another dataset or a custom one can be used

```
X, y = load_iris(return_X_y=True)
```

Define the HyperparameterSpace, you can use the default hyperparameters for the machine learning model that you want to optimize. In this case we use the default hyperparameters for a DecisionTreeClassifier. Another dataset or a custom one can be used

```
hyperparam_space = HyperparameterSpace.get_default_hyperparameter_
↪space(DecisionTreeClassifier)
```

We use the default TreeOptimizer class to optimize a decision tree classifier.

```
opt = Optimizer(estimator_class=DecisionTreeClassifier, features=X, labels=y,
                hyperparam_space=hyperparam_space, folder="Search_space_example")
```

To optimizer the classifier we need to call the optimize_clf method. The first argument is the number of generations and the second is the number of individuals in each generation.

```
clf = opt.optimize_clf(10, 10)
```

```
Genetic execution:   0%|          | 0/11 [00:00<?, ?it/s]/home/docs/checkouts/
↪readthedocs.org/user_builds/mloptimizer/envs/dev/lib/python3.10/site-packages/deap/
↪creator.py:185: RuntimeWarning:

A class named 'FitnessMax' has already been created and it will be overwritten. Consider␣
↪deleting previous creation of that class or rename it.

/home/docs/checkouts/readthedocs.org/user_builds/mloptimizer/envs/dev/lib/python3.10/
↪site-packages/deap/creator.py:185: RuntimeWarning:

A class named 'Individual' has already been created and it will be overwritten. Consider␣
↪deleting previous creation of that class or rename it.


Genetic execution:  45%|     | 5/11 [00:00<00:00, 48.42it/s]
Genetic execution:  91%| | 10/11 [00:00<00:00, 48.05it/s]
Genetic execution: 100%|| 11/11 [00:00<00:00, 44.04it/s]
```

Following we can generate the plot of the search space

```
population_df = opt.runs[-1].population_2_df()
param_names = list(opt.hyperparam_space.evolvable_hyperparams.keys())
param_names.append("fitness")
df = population_df[param_names]
g_search_space = plotly_search_space(df, param_names)
plotly.io.show(g_search_space)
```

At the end of the evolution the graph is saved as an html at the path:

```
print(opt.tracker.graphics_path)
print(os.listdir(opt.tracker.graphics_path))
```

```
Search_space_example/20240406_161633_Tracker/graphics
['logbook.html', 'search_space.html']
```

The data to generate the graph is available at the path:

```
print(opt.tracker.results_path)
print(os.listdir(opt.tracker.results_path))

del opt
```

```
Search_space_example/20240406_161633_Tracker/results
['populations.csv', 'logbook.csv']
```

**Total running time of the script:** (0 minutes 1.896 seconds)

## 6.3 Quickstart example

Quick example of use of the library to optimize a decision tree classifier. Firstly, we import the necessary libraries to get data and plot the results.

```
from mloptimizer.core import Optimizer
from mloptimizer.hyperparams import HyperparameterSpace
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_iris
```

Load the iris dataset to obtain a vector of features X and a vector of labels y. Another dataset or a custom one can be used

```
X, y = load_iris(return_X_y=True)
```

Split the dataset into training and test sets

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2)
```

Define the HyperparameterSpace, you can use the default hyperparameters for the machine learning model that you want to optimize. In this case we use the default hyperparameters for a DecisionTreeClassifier. Another dataset or a custom one can be used

```
hyperparam_space = HyperparameterSpace.get_default_hyperparameter_
↪space(DecisionTreeClassifier)
```

The TreeOptimizer class is the main wrapper for the optimization of a Decision Tree Classifier. The first argument is the vector of features, the second is the vector of labels and the third (if provided) is the name of the folder where the results of mloptimizer Optimizers are saved. The default value for this folder is “Optimizer”

```
opt = Optimizer(estimator_class=DecisionTreeClassifier, features=X_train, labels=y_train,
                hyperparam_space=hyperparam_space, folder="Optimizer")
```

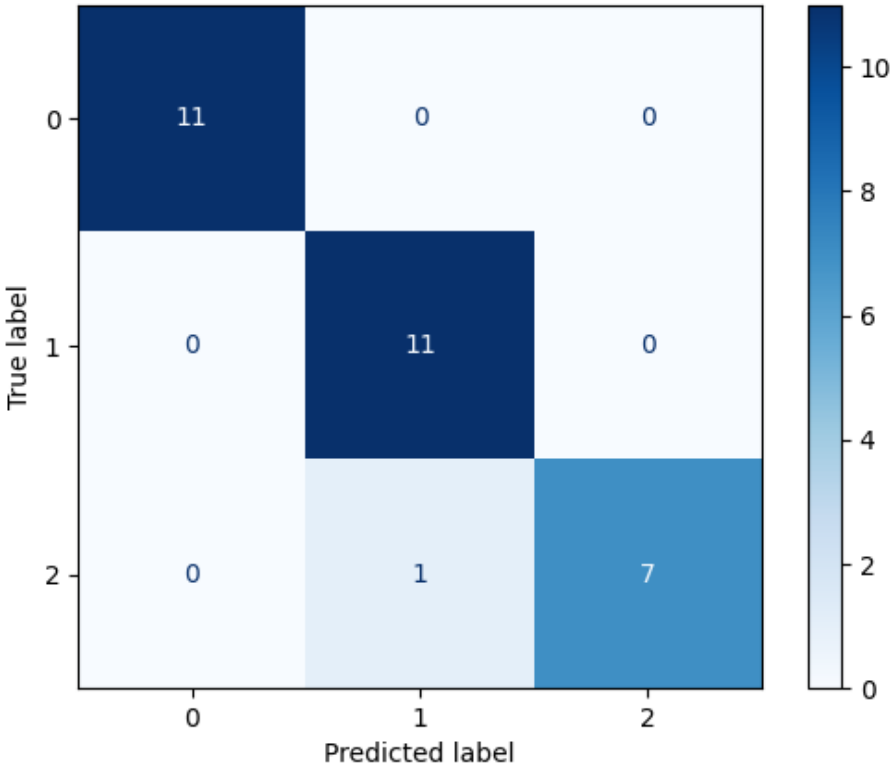
To optimizer the classifier we need to call the optimize_clf method. The first argument is the number of generations and the second is the number of individuals in each generation. The method returns the best classifier with the best hyperparameters found.

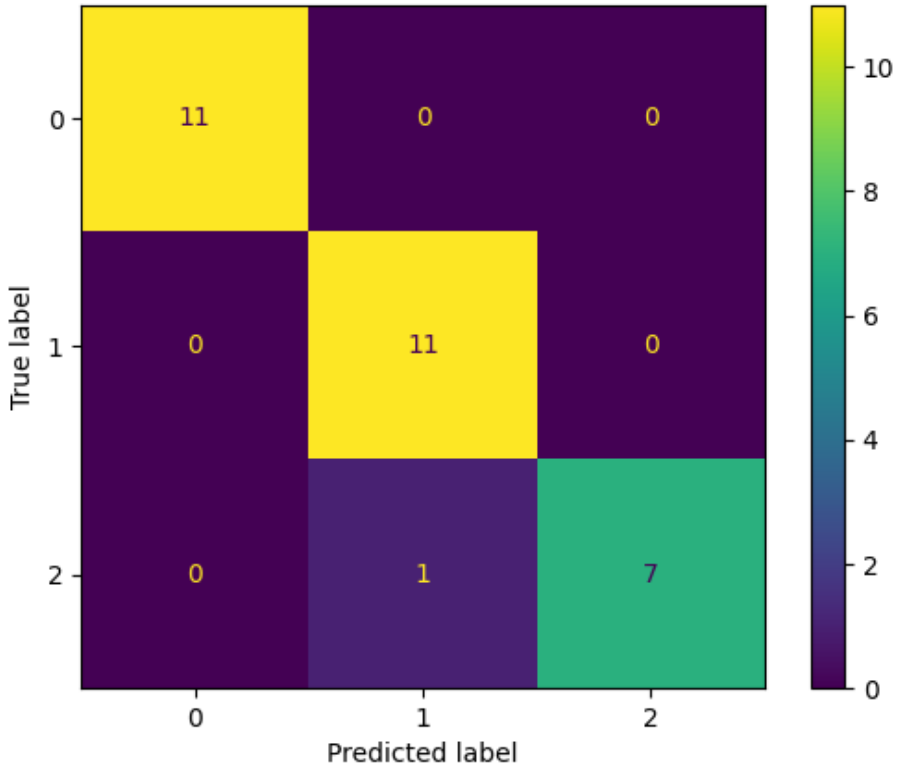
```
clf = opt.optimize_clf(10, 10)

print(clf)
```

```
Genetic execution:   0%|          | 0/11 [00:00<?, ?it/s]/home/docs/checkouts/
↪readthedocs.org/user_builds/mloptimizer/envs/dev/lib/python3.10/site-packages/deap/
↪creator.py:185: RuntimeWarning:

A class named 'FitnessMax' has already been created and it will be overwritten. Consider␣
↪deleting previous creation of that class or rename it.

/home/docs/checkouts/readthedocs.org/user_builds/mloptimizer/envs/dev/lib/python3.10/
↪site-packages/deap/creator.py:185: RuntimeWarning:

A class named 'Individual' has already been created and it will be overwritten. Consider␣
↪deleting previous creation of that class or rename it.


Genetic execution:  55%|    | 6/11 [00:00<00:00, 53.52it/s]
Genetic execution: 100%|| 11/11 [00:00<00:00, 46.16it/s]
DecisionTreeClassifier(ccp_alpha=0.00256, max_depth=19,
                       min_impurity_decrease=0.063, min_samples_leaf=20,
                       min_samples_split=38, random_state=402428)
```

Train the classifier with the best hyperparameters found Show the classification report and the confusion matrix

```
from sklearn.metrics import classification_report, confusion_matrix, \
    ConfusionMatrixDisplay
import matplotlib.pyplot as plt

clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
cm = confusion_matrix(y_test, y_pred)
print(classification_report(y_test, y_pred))
disp = ConfusionMatrixDisplay.from_predictions(
    y_test, y_pred, display_labels=clf.classes_,
    cmap=plt.cm.Blues
)
disp.plot()
plt.show()

del opt
```


11
0
0
0
11
0
0
1
7
0
1
2
True label
Predicted label
0
2
4
6
8
10




```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        11
           1       0.92      1.00      0.96        11
           2       1.00      0.88      0.93         8

    accuracy                           0.97        30
   macro avg       0.97      0.96      0.96        30
weighted avg       0.97      0.97      0.97        30
```

**Total running time of the script:** (0 minutes 0.604 seconds)

CHAPTER

SEVEN

# MLOPTIMIZER UI

This page is here, to provide some content for the use of the MLOptimizer UI. (Work in progress)

CHAPTER

# EIGHT

# API REFERENCE

This page contains auto-generated API reference documentation[1].

## 8.1 mloptimizer

### 8.1.1 Subpackages

#### mloptimizer.aux

#### Submodules

#### mloptimizer.aux.alg_wrapper

#### Module Contents

#### Classes

| | |
|---|---|
| *CustomXGBClassifier* | A class to wrap the xgboost classifier. |

#### Functions

| |
|---|
| *generate_model*([learning_rate, layer_1, layer_2, ...]) |

**class** mloptimizer.aux.alg_wrapper.**CustomXGBClassifier**(*base_score=0.5*, *booster='gbtree'*, *eval_metric='auc'*, *eta=0.077*, *gamma=18*, *subsample=0.728*, *colsample_bylevel=1*, *colsample_bytree=0.46*, *max_delta_step=0*, *max_depth=7*, *min_child_weight=1*, *seed=1*, *alpha=0*, *reg_lambda=1*, *scale_pos_weight=4.43*, *obj=None*, *feval=None*, *num_boost_round=50*)

[1] Created with sphinx-autoapi

Bases: sklearn.base.BaseEstimator

A class to wrap the xgboost classifier.

**base_score**

The initial prediction score of all instances, global bias.

**Type**
float, optional (default=0.5)

**booster**

Which booster to use, can be gbtree, gblinear or dart; gbtree and dart use tree based models while gblinear uses linear functions.

**Type**
string, optional (default="gbtree")

**eval_metric**

Evaluation metrics for validation data, a default metric will be assigned according to objective (rmse for regression, and error for classification, mean average precision for ranking).

**Type**
string, optional (default="auc")

**eta**

Step size shrinkage used in update to prevent overfitting.

**Type**
float, optional (default=0.077)

**gamma**

Minimum loss reduction required to make a further partition on a leaf node of the tree.

**Type**
float, optional (default=18)

**subsample**

Subsample ratio of the training instance.

**Type**
float, optional (default=0.728)

**colsample_bylevel**

Subsample ratio of columns for each split, in each level.

**Type**
float, optional (default=1)

**colsample_bytree**

Subsample ratio of columns when constructing each tree.

**Type**
float, optional (default=0.46)

**max_delta_step**

Maximum delta step we allow each tree's weight estimation to be.

**Type**
int, optional (default=0)

**max_depth**

Maximum depth of a tree.

**Type**
int, optional (default=7)

**min_child_weight**

Minimum sum of instance weight(hessian) needed in a child.

**Type**
int, optional (default=1)

**seed**

Random number seed.

**Type**
int, optional (default=1)

**alpha**

L1 regularization term on weights.

**Type**
float, optional (default=0)

**reg_lambda**

L2 regularization term on weights.

**Type**
float, optional (default=1)

**scale_pos_weight**

Balancing of positive and negative weights.

**Type**
float, optional (default=4.43)

**obj**

Customized objective function.

**Type**
callable, optional (default=None)

**feval**

Customized evaluation function.

**Type**
callable, optional (default=None)

**num_boost_round**

Number of boosting iterations.

**Type**
int, optional (default=50)

**fit**(*X*, *y*)

Fit the model according to the given training data.

**Parameters**

- **X** (*array-like of shape (n_samples, n_features)*) – The training input samples.

- **y** (*array-like of shape (n_samples,)*) – The target values (class labels in classification, real numbers in regression).

**Returns**
**self** – Returns self.

**Return type**
object

**predict**(*X*)

Predict class labels for samples in X.

**Parameters**
**X** (*array-like of shape (n_samples, n_features)*) – The input samples.

**Returns**
**preds** – The predicted classes.

**Return type**
array-like of shape (n_samples,)

**predict_proba**(*X*)

Predict class probabilities for samples in X.

**Parameters**
**X** (*array-like of shape (n_samples, n_features)*) – The input samples.

**Returns**
**p** – The predicted probabilities.

**Return type**
array-like of shape (n_samples,)

**predict_z**(*X*)

Predict z values for samples in X.

**Parameters**
**X** (*array-like of shape (n_samples, n_features)*) – The input samples.

**Returns**
**zs** – The predicted z values.

**Return type**
array-like of shape (n_samples,)

mloptimizer.aux.alg_wrapper.**generate_model**(*learning_rate=0.01*, *layer_1=100*, *layer_2=50*, *dropout_rate_1=0*, *dropout_rate_2=0*)

## mloptimizer.aux.plots

### Module Contents

### Functions

| | |
|---|---|
| *logbook_to_pandas*(logbook) | Function to transform a deap logbook to a pandas dataframe. |
| *plotly_logbook*(logbook, population) | Generate plotly figure from logbook. Evolution of fitness and population. |
| *plot_logbook*(logbook) | Generate sns figure from logbook. Evolution of fitness and population. |
| *plotly_search_space*(populations_df[, features]) | Generate plotly figure from populations dataframe and features. Search space. |
| *plot_search_space*(populations_df[, height, s]) | Generate sns figure from populations dataframe and features. Search space. |

mloptimizer.aux.plots.**logbook_to_pandas**(*logbook*)

Function to transform a deap logbook to a pandas dataframe.

**Parameters**
**logbook** (*deap.tools.Logbook*) – The logbook to transform

**Returns**
**df** – The logbook as a pandas dataframe

**Return type**
pd.DataFrame

mloptimizer.aux.plots.**plotly_logbook**(*logbook*, *population*)

Generate plotly figure from logbook. Evolution of fitness and population.

**Parameters**

- **logbook** (*deap.tools.Logbook*) – The logbook to plot
- **population** (*pd.DataFrame*) – The population to plot

**Returns**
**fig** – The figure

**Return type**
plotly.graph_objects.Figure

mloptimizer.aux.plots.**plot_logbook**(*logbook*)

Generate sns figure from logbook. Evolution of fitness and population.

**Parameters**
**logbook** (*deap.tools.Logbook*) – The logbook to plot

**Returns**
**fig** – The figure

**Return type**
plotly.graph_objects.Figure

mloptimizer.aux.plots.**plotly_search_space**(*populations_df: pandas.DataFrame*, *features: list = None*)

Generate plotly figure from populations dataframe and features. Search space.

**Parameters**

- **populations_df** (*pd.DataFrame*) – The dataframe with the population
- **features** (*list*) – The features to plot

**Returns**
**fig** – The figure

**Return type**
plotly.graph_objects.Figure

mloptimizer.aux.plots.**plot_search_space**(*populations_df: pandas.DataFrame*, *height=2*, *s=25*)

Generate sns figure from populations dataframe and features. Search space.

**Parameters**

- **populations_df** (*pd.DataFrame*) – The dataframe with the population
- **height** (*int*) – The height of the figure
- **s** (*int*) – The size of the points

**Returns**
**fig** – The figure

**Return type**
plotly.graph_objects.Figure

## mloptimizer.aux.tracker

### Module Contents

### Classes

| | |
|---|---|
| *Tracker* | Tracker class for logging and tracking the optimization process. |

class mloptimizer.aux.tracker.**Tracker**(*name*, *folder=os.curdir*, *log_file='mloptimizer.log'*, *use_mlflow=False*, *use_parallel=False*)

Tracker class for logging and tracking the optimization process.

**Parameters**

- **name** (*str*) – Name of the optimization process.
- **folder** (*str*) – Folder where the optimization process will be stored.
- **log_file** (*str*) – Name of the log file.
- **use_mlflow** (*bool*) – If True, the optimization process will be tracked using MLFlow.

**start_optimization**(*opt_class*, *generations: int*)

Start the optimization process.

**Parameters**

- **opt_class** (*str*) – Name of the optimization class.
- **generations** (*int*) – Number of generations for the optimization process.

**start_checkpoint**(*opt_run_folder_name*)

Start a checkpoint for the optimization process.

**Parameters**
**opt_run_folder_name** (*str*) – Name of the folder where the checkpoint will be stored. (not the full path)

**log_clfs**(*classifiers_list: list*, *generation: int*, *fitness_list: list[int]*)

**log_evaluation**(*classifier*, *metrics*)

**load_checkpoint**(*checkpoint*)

**write_logbook_file**(*logbook*, *filename=None*)
Method to write the logbook to a csv file

**Parameters**
- **logbook** (*Logbook*) – logbook of the optimization process
- **filename** (*str, optional (default=None)*) – filename to save the logbook

**write_population_file**(*populations: pandas.DataFrame*, *filename=None*)
Method to write the population to a csv file

**Parameters**
- **populations** (*pd.DataFrame*) – population of the optimization process
- **filename** (*str, optional (default=None)*) – filename to save the population

**start_progress_file**(*gen: int*)

**append_progress_file**(*gen: int*, *ngen: int*, *c: int*, *evaluations_pending: int*, *ind_formatted*, *fit*)

## mloptimizer.aux.utils

### Module Contents

### Functions

| | |
|---|---|
| *init_logger*([filename, log_path, debug]) | Initializes a logger and returns it. |
| *create_optimization_folder*(folder) | Creates a folder to save the results of the optimization. |

mloptimizer.aux.utils.**init_logger**(*filename='mloptimizer.log'*, *log_path='.'*, *debug=False*)
Initializes a logger and returns it.

**Parameters**
- **filename** (*str, optional*) – The name of the log file. The default is 'mloptimizer.log'.
- **log_path** (*str, optional*) – The path of the log file. The default is ".".
- **debug** (*bool, optional*) – Activate debug level. The default is False.

**Returns**
**custom_logger** – The logger.

**Return type**
logging.Logger

mloptimizer.aux.utils.**create_optimization_folder**(*folder*)

Creates a folder to save the results of the optimization.

**Parameters**
**folder** (*str*) – The path of the folder to create.

**Returns**
**folder** – The path of the folder created.

**Return type**
str

## Package Contents

## Classes

| | |
|---|---|
| *Tracker* | Tracker class for logging and tracking the optimization process. |

class mloptimizer.aux.**Tracker**(*name*, *folder=os.curdir*, *log_file='mloptimizer.log'*, *use_mlflow=False*, *use_parallel=False*)

Tracker class for logging and tracking the optimization process.

**Parameters**

- **name** (*str*) – Name of the optimization process.
- **folder** (*str*) – Folder where the optimization process will be stored.
- **log_file** (*str*) – Name of the log file.
- **use_mlflow** (*bool*) – If True, the optimization process will be tracked using MLFlow.

**start_optimization**(*opt_class*, *generations: int*)

Start the optimization process.

**Parameters**

- **opt_class** (*str*) – Name of the optimization class.
- **generations** (*int*) – Number of generations for the optimization process.

**start_checkpoint**(*opt_run_folder_name*)

Start a checkpoint for the optimization process.

**Parameters**
**opt_run_folder_name** (*str*) – Name of the folder where the checkpoint will be stored. (not the full path)

**log_clfs**(*classifiers_list: list*, *generation: int*, *fitness_list: list[int]*)

**log_evaluation**(*classifier*, *metrics*)

**load_checkpoint**(*checkpoint*)

**write_logbook_file**(*logbook*, *filename=None*)

Method to write the logbook to a csv file

**Parameters**

- **logbook** (*Logbook*) – logbook of the optimization process
- **filename** (*str, optional (default=None)*) – filename to save the logbook

**write_population_file**(*populations: pandas.DataFrame*, *filename=None*)

Method to write the population to a csv file

**Parameters**

- **populations** (*pd.DataFrame*) – population of the optimization process
- **filename** (*str, optional (default=None)*) – filename to save the population

**start_progress_file**(*gen: int*)

**append_progress_file**(*gen: int*, *ngen: int*, *c: int*, *evaluations_pending: int*, *ind_formatted*, *fit*)

## mloptimizer.core

### Submodules

## mloptimizer.core.base

### Module Contents

### Classes

| | |
|---|---|
| *Optimizer* | Base class for the optimization of a classifier |

**class** mloptimizer.core.base.**Optimizer**(*estimator_class*, *features: numpy.array*, *labels: numpy.array*, *folder=os.curdir*, *log_file='mloptimizer.log'*, *hyperparam_space:* mloptimizer.hyperparams.HyperparameterSpace *= None*, *eval_function=train_score*, *fitness_score='accuracy'*, *metrics=None*, *seed=random.randint(0, 1000000)*, *use_parallel=False*, *use_mlflow=False*)

Base class for the optimization of a classifier

**estimator_class**

class of the classifier

**Type**
class

**features**

np.array with the features

**Type**
np.array

`labels`
: np.array with the labels

  **Type**
  : np.array

`hyperparam_space`
: object with the hyperparameter space: fixed and evolvable hyperparams

  **Type**
  : *HyperparameterSpace*

`evaluator`
: object to evaluate the classifier

  **Type**
  : *Evaluator*

`eval_dict`
: dictionary with the evaluation of the individuals

  **Type**
  : dict

`populations`
: list of populations

  **Type**
  : list

`logbook`
: list of logbook

  **Type**
  : list

`seed`
: seed for the random functions

  **Type**
  : int

`use_parallel`
: flag to use parallel processing

  **Type**
  : bool

`use_mlflow`
: flag to use mlflow

  **Type**
  : bool

`set_mlopt_seed`(*seed*)
: Method to set the seed for the random functions

  **Parameters**
  : **seed** (*int*) – seed for the random functions

**static get_subclasses**(*my_class*)

Method to get all the subclasses of a class (in this case use to get all the classifiers that can be optimized).

**Parameters**
**my_class** (*class*) – class to get the subclasses

**Returns**
list of subclasses

**Return type**
list

**get_clf**(*individual*)

**optimize_clf**(*population_size: int = 10*, *generations: int = 3*, *cxpb=0.5*, *mutpb=0.5*, *tournsize=4*, *indpb=0.5*, *n_elites=10*, *checkpoint: str = None*, *opt_run_folder_name: str = None*) → object

Method to optimize the classifier. It uses the custom_ea_simple method to optimize the classifier.

**Parameters**

- **population_size** (*int, optional (default=10)*) – number of individuals in each generation
- **generations** (*int, optional (default=3)*) – number of generations
- **cxpb** (*float, optional (default=0.5)*) – crossover probability
- **mutpb** (*float, optional (default=0.5)*) – mutation probability
- **tournsize** (*int, optional (default=4)*) – number of individuals to select in the tournament
- **indpb** (*float, optional (default=0.5)*) – independent probability for each attribute to be mutated
- **n_elites** (*int, optional (default=10)*) – number of elites to keep in the next generation
- **checkpoint** (*str, optional (default=None)*) – path to the checkpoint file
- **opt_run_folder_name** (*str, optional (default=None)*) – name of the folder where the execution will be saved

**Returns**
**clf** – classifier with the best hyperparams

**Return type**
classifier

## mloptimizer.core.keras

### Module Contents

### Classes

| | |
|---|---|
| *KerasClassifierOptimizer* | Class for the optimization of a gradient boosting classifier from keras.wrappers.scikit_learn.KerasClassifier. |

class mloptimizer.core.keras.**KerasClassifierOptimizer**(*estimator_class*, *features: numpy.array*, *labels: numpy.array*, *folder=os.curdir*, *log_file='mloptimizer.log'*, *hyperparam_space:* mloptimizer.hyperparams.HyperparameterSpace = *None*, *eval_function=train_score*, *fitness_score='accuracy'*, *metrics=None*, *seed=random.randint(0, 1000000)*, *use_parallel=False*, *use_mlflow=False*)

Bases: *mloptimizer.core.Optimizer*

Class for the optimization of a gradient boosting classifier from keras.wrappers.scikit_learn.KerasClassifier. It inherits from BaseOptimizer.

**static get_default_hyperparams**()

**get_clf**(*individual*)

## Package Contents

## Classes

| | |
|---|---|
| *Optimizer* | Base class for the optimization of a classifier |
| *KerasClassifierOptimizer* | Class for the optimization of a gradient boosting classifier from keras.wrappers.scikit_learn.KerasClassifier. |

class mloptimizer.core.**Optimizer**(*estimator_class*, *features: numpy.array*, *labels: numpy.array*, *folder=os.curdir*, *log_file='mloptimizer.log'*, *hyperparam_space:* mloptimizer.hyperparams.HyperparameterSpace = *None*, *eval_function=train_score*, *fitness_score='accuracy'*, *metrics=None*, *seed=random.randint(0, 1000000)*, *use_parallel=False*, *use_mlflow=False*)

Base class for the optimization of a classifier

**estimator_class**

class of the classifier

**Type**
class

**features**

np.array with the features

**Type**
np.array

**labels**

np.array with the labels

**Type**
np.array

**hyperparam_space**

object with the hyperparameter space: fixed and evolvable hyperparams

**Type**
*HyperparameterSpace*

**evaluator**

object to evaluate the classifier

**Type**
*Evaluator*

**eval_dict**

dictionary with the evaluation of the individuals

**Type**
dict

**populations**

list of populations

**Type**
list

**logbook**

list of logbook

**Type**
list

**seed**

seed for the random functions

**Type**
int

**use_parallel**

flag to use parallel processing

**Type**
bool

**use_mlflow**

flag to use mlflow

**Type**
bool

**set_mlopt_seed**(*seed*)

Method to set the seed for the random functions

**Parameters**
**seed** (*int*) – seed for the random functions

**static get_subclasses**(*my_class*)

Method to get all the subclasses of a class (in this case use to get all the classifiers that can be optimized).

**Parameters**
**my_class** (*class*) – class to get the subclasses

**Returns**
list of subclasses

**Return type**
list

`get_clf`(*individual*)

`optimize_clf`(*population_size: int = 10*, *generations: int = 3*, *cxpb=0.5*, *mutpb=0.5*, *tournsize=4*, *indpb=0.5*, *n_elites=10*, *checkpoint: str = None*, *opt_run_folder_name: str = None*) → object

Method to optimize the classifier. It uses the custom_ea_simple method to optimize the classifier.

**Parameters**

- **population_size** (*int, optional (default=10)*) – number of individuals in each generation
- **generations** (*int, optional (default=3)*) – number of generations
- **cxpb** (*float, optional (default=0.5)*) – crossover probability
- **mutpb** (*float, optional (default=0.5)*) – mutation probability
- **tournsize** (*int, optional (default=4)*) – number of individuals to select in the tournament
- **indpb** (*float, optional (default=0.5)*) – independent probability for each attribute to be mutated
- **n_elites** (*int, optional (default=10)*) – number of elites to keep in the next generation
- **checkpoint** (*str, optional (default=None)*) – path to the checkpoint file
- **opt_run_folder_name** (*str, optional (default=None)*) – name of the folder where the execution will be saved

**Returns**
**clf** – classifier with the best hyperparams

**Return type**
classifier

`class mloptimizer.core.KerasClassifierOptimizer`(*estimator_class*, *features: numpy.array*, *labels: numpy.array*, *folder=os.curdir*, *log_file='mloptimizer.log'*, *hyperparam_space: mloptimizer.hyperparams.HyperparameterSpace = None*, *eval_function=train_score*, *fitness_score='accuracy'*, *metrics=None*, *seed=random.randint(0, 1000000)*, *use_parallel=False*, *use_mlflow=False*)

Bases: *mloptimizer.core.Optimizer*

Class for the optimization of a gradient boosting classifier from keras.wrappers.scikit_learn.KerasClassifier. It inherits from BaseOptimizer.

`static get_default_hyperparams`()

`get_clf`(*individual*)

## mloptimizer.evaluation

### Submodules

## mloptimizer.evaluation.evaluator

### Module Contents

### Classes

| | |
|---|---|
| *Evaluator* | Evaluator class to evaluate the performance of a classifier |

**class** mloptimizer.evaluation.evaluator.**Evaluator**(*features: numpy.array*, *labels: numpy.array*, *eval_function*, *fitness_score='accuracy'*, *metrics=None*, *tracker:* mloptimizer.aux.Tracker *= None*, *individual_utils=None*)

Evaluator class to evaluate the performance of a classifier

**Parameters**

- **features** (*array-like*) – The features to use to evaluate the classifier
- **labels** (*array-like*) – The labels to use to evaluate the classifier
- **eval_function** (*function*) – The evaluation function to use to evaluate the performance of the classifier
- **fitness_score** (*str*) – The fitness score to use to evaluate the performance of the classifier
- **metrics** (*dict*) – The metrics to use to evaluate the performance of the classifier Dictionary of the form {"metric_name": metric_function}
- **tracker** (Tracker) – The tracker to use to log the evaluations
- **individual_utils** (IndividualUtils) – The individual utils to use to get the classifier from the individual

**evaluate**(*clf*, *features*, *labels*)

Evaluate the performance of a classifier

**Parameters**

- **clf** (*object*) – The classifier to evaluate
- **features** (*array-like*) – The features to use to evaluate the classifier
- **labels** (*array-like*) – The labels to use to evaluate the classifier

**Returns**

**metrics** – Dictionary of the form {"metric_name": metric_value}

**Return type**

dict

**evaluate_individual**(*individual*)

## mloptimizer.evaluation.model_evaluation

### Module Contents

### Functions

| | |
|---|---|
| score_metrics(labels, predictions, metrics) | |
| train_score(features, labels, clf, metrics) | Trains the classifier with the features and labels. |
| train_test_score(features, labels, clf, metrics[, ...]) | Trains the classifier with the train set features and labels, |
| kfold_score(features, labels, clf, metrics[, ...]) | Evaluates the classifier using K-Fold cross-validation. |
| kfold_stratified_score(features, labels, clf, metrics) | Computes KFold cross validation score using n_splits folds. |
| temporal_kfold_score(features, labels, clf, metrics[, ...]) | Computes KFold cross validation score using n_splits folds. |

mloptimizer.evaluation.model_evaluation.**score_metrics**(*labels*, *predictions*, *metrics*)

mloptimizer.evaluation.model_evaluation.**train_score**(*features*, *labels*, *clf*, *metrics*)

Trains the classifier with the features and labels.

**Parameters**

- **features** (*list*) – List of features
- **labels** (*list*) – List of labels
- **clf** (*object*) – classifier with methods fit, predict and score
- **metrics** (*dict*) – dictionary with metrics to be used keys are the name of the metric and values are the metric function

**Returns**

**metrics_output** – dictionary with the metrics over the train set

**Return type**

dict

mloptimizer.evaluation.model_evaluation.**train_test_score**(*features*, *labels*, *clf*, *metrics*, *test_size=0.2*, *random_state=None*)

Trains the classifier with the train set features and labels, then uses the test features and labels to create score.

**Parameters**

- **features** (*list*) – List of features
- **labels** (*list*) – List of labels
- **clf** (*object*) – Classifier with methods fit, predict, and score
- **metrics** (*dict*) – dictionary with metrics to be used keys are the name of the metric and values are the metric function
- **test_size** (*float, optional*) – Proportion of the dataset to include in the test split
- **random_state** (*int, optional*) – Controls the shuffling applied to the data before applying the split

**Returns**
**metrics_output** – dictionary with the metrics over the test set

**Return type**
dict

mloptimizer.evaluation.model_evaluation.**kfold_score**(*features*, *labels*, *clf*, *metrics*, *n_splits=5*, *random_state=None*)

Evaluates the classifier using K-Fold cross-validation.

**Parameters**

- **features** (*array-like*) – Array of features
- **labels** (*array-like*) – Array of labels
- **clf** (*object*) – Classifier with methods fit and predict
- **metrics** (*dict*) – dictionary with metrics to be used keys are the name of the metric and values are the metric function
- **n_splits** (*int, optional*) – Number of folds. Must be at least 2
- **random_state** (*int, optional*) – Controls the randomness of the fold assignment

**Returns**
**average_metrics** – mean score among k-folds test splits

**Return type**
dict

mloptimizer.evaluation.model_evaluation.**kfold_stratified_score**(*features*, *labels*, *clf*, *metrics*, *n_splits=4*, *random_state=None*)

Computes KFold cross validation score using n_splits folds. It uses the features and labels to train the k-folds. Uses a stratified KFold split. The score_function is the one used to score each k-fold.

**Parameters**

- **features** (*list*) – List of features
- **labels** (*list*) – List of labels
- **clf** (*object*) – classifier with methods fit, predict and score
- **n_splits** (*int*) – number of splits
- **metrics** (*dict*) – dictionary with metrics to be used keys are the name of the metric and values are the metric function
- **random_state** (*int*) – random state for the stratified kfold

**Returns**
**average_metrics** – mean score among k-folds test splits

**Return type**
dict

mloptimizer.evaluation.model_evaluation.**temporal_kfold_score**(*features*, *labels*, *clf*, *metrics*, *n_splits=4*)

Computes KFold cross validation score using n_splits folds. It uses the features and labels to train the k-folds. Uses a temporal KFold split. The score_function is the one used to score each k-fold.

**Parameters**

- **features** (*list*) – List of features

- **labels** (*list*) – List of labels
- **clf** (*object*) – classifier with methods fit, predict and score
- **n_splits** (*int*) – number of splits
- **metrics** (*dict*) – dictionary with metrics to be used keys are the name of the metric and values are the metric function

**Returns**
**average_metrics** – mean score among k-folds test splits

**Return type**
dict

## Package Contents

## Classes

| | |
|---|---|
| *Evaluator* | Evaluator class to evaluate the performance of a classifier |

## Functions

| | |
|---|---|
| *kfold_stratified_score*(features, labels, clf, metrics) | Computes KFold cross validation score using n_splits folds. |
| *temporal_kfold_score*(features, labels, clf, metrics[, ...]) | Computes KFold cross validation score using n_splits folds. |
| *train_score*(features, labels, clf, metrics) | Trains the classifier with the features and labels. |
| *train_test_score*(features, labels, clf, metrics[, ...]) | Trains the classifier with the train set features and labels, |
| *kfold_score*(features, labels, clf, metrics[, ...]) | Evaluates the classifier using K-Fold cross-validation. |

mloptimizer.evaluation.**kfold_stratified_score**(*features*, *labels*, *clf*, *metrics*, *n_splits=4*, *random_state=None*)

Computes KFold cross validation score using n_splits folds. It uses the features and labels to train the k-folds. Uses a stratified KFold split. The score_function is the one used to score each k-fold.

**Parameters**

- **features** (*list*) – List of features
- **labels** (*list*) – List of labels
- **clf** (*object*) – classifier with methods fit, predict and score
- **n_splits** (*int*) – number of splits
- **metrics** (*dict*) – dictionary with metrics to be used keys are the name of the metric and values are the metric function
- **random_state** (*int*) – random state for the stratified kfold

**Returns**
**average_metrics** – mean score among k-folds test splits

**Return type**
dict

mloptimizer.evaluation.**temporal_kfold_score**(*features*, *labels*, *clf*, *metrics*, *n_splits=4*)

Computes KFold cross validation score using n_splits folds. It uses the features and labels to train the k-folds. Uses a temporal KFold split. The score_function is the one used to score each k-fold.

**Parameters**

- **features** (*list*) – List of features
- **labels** (*list*) – List of labels
- **clf** (*object*) – classifier with methods fit, predict and score
- **n_splits** (*int*) – number of splits
- **metrics** (*dict*) – dictionary with metrics to be used keys are the name of the metric and values are the metric function

**Returns**
**average_metrics** – mean score among k-folds test splits

**Return type**
dict

mloptimizer.evaluation.**train_score**(*features*, *labels*, *clf*, *metrics*)

Trains the classifier with the features and labels.

**Parameters**

- **features** (*list*) – List of features
- **labels** (*list*) – List of labels
- **clf** (*object*) – classifier with methods fit, predict and score
- **metrics** (*dict*) – dictionary with metrics to be used keys are the name of the metric and values are the metric function

**Returns**
**metrics_output** – dictionary with the metrics over the train set

**Return type**
dict

mloptimizer.evaluation.**train_test_score**(*features*, *labels*, *clf*, *metrics*, *test_size=0.2*, *random_state=None*)

Trains the classifier with the train set features and labels, then uses the test features and labels to create score.

**Parameters**

- **features** (*list*) – List of features
- **labels** (*list*) – List of labels
- **clf** (*object*) – Classifier with methods fit, predict, and score
- **metrics** (*dict*) – dictionary with metrics to be used keys are the name of the metric and values are the metric function
- **test_size** (*float, optional*) – Proportion of the dataset to include in the test split
- **random_state** (*int, optional*) – Controls the shuffling applied to the data before applying the split

**Returns**
**metrics_output** – dictionary with the metrics over the test set

**Return type**
dict

mloptimizer.evaluation.**kfold_score**(*features*, *labels*, *clf*, *metrics*, *n_splits=5*, *random_state=None*)

Evaluates the classifier using K-Fold cross-validation.

**Parameters**

- **features** (*array-like*) – Array of features
- **labels** (*array-like*) – Array of labels
- **clf** (*object*) – Classifier with methods fit and predict
- **metrics** (*dict*) – dictionary with metrics to be used keys are the name of the metric and values are the metric function
- **n_splits** (*int, optional*) – Number of folds. Must be at least 2
- **random_state** (*int, optional*) – Controls the randomness of the fold assignment

**Returns**
**average_metrics** – mean score among k-folds test splits

**Return type**
dict

class mloptimizer.evaluation.**Evaluator**(*features: numpy.array*, *labels: numpy.array*, *eval_function*, *fitness_score='accuracy'*, *metrics=None*, *tracker: mloptimizer.aux.Tracker = None*, *individual_utils=None*)

Evaluator class to evaluate the performance of a classifier

**Parameters**

- **features** (*array-like*) – The features to use to evaluate the classifier
- **labels** (*array-like*) – The labels to use to evaluate the classifier
- **eval_function** (*function*) – The evaluation function to use to evaluate the performance of the classifier
- **fitness_score** (*str*) – The fitness score to use to evaluate the performance of the classifier
- **metrics** (*dict*) – The metrics to use to evaluate the performance of the classifier Dictionary of the form {"metric_name": metric_function}
- **tracker** (Tracker) – The tracker to use to log the evaluations
- **individual_utils** (IndividualUtils) – The individual utils to use to get the classifier from the individual

**evaluate**(*clf*, *features*, *labels*)

Evaluate the performance of a classifier

**Parameters**

- **clf** (*object*) – The classifier to evaluate
- **features** (*array-like*) – The features to use to evaluate the classifier
- **labels** (*array-like*) – The labels to use to evaluate the classifier

**Returns**
**metrics** – Dictionary of the form {"metric_name": metric_value}

**Return type**
dict

`evaluate_individual`(*individual*)

## mloptimizer.genetic

### Submodules

## mloptimizer.genetic.deapoptimizer

### Module Contents

### Classes

| |
|---|
| *DeapOptimizer* |

`class mloptimizer.genetic.deapoptimizer.DeapOptimizer`(*hyperparam_space:* mloptimizer.hyperparams.HyperparameterSpace = *None*, *use_parallel=False*, *seed=None*)

`init_individual`(*pcls*)
Method to create an individual

**Parameters**
**pcls** (*class*) – class of the individual

**Returns**
**ind** – individual

**Return type**
individual

`individual2dict`(*individual*)
Method to convert an individual to a dictionary of hyperparams

**Parameters**
**individual** (*individual*) – individual to convert

**Returns**
**individual_dict** – dictionary of hyperparams

**Return type**
dict

`setup`()
Method to set the parameters for the optimization.

mloptimizer.genetic.garunner

## Module Contents

## Classes

| |
|---|
| *GeneticAlgorithmRunner* |

class mloptimizer.genetic.garunner.**GeneticAlgorithmRunner**(*deap_optimizer:* mloptimizer.genetic.DeapOptimizer, *tracker:* mloptimizer.aux.Tracker, *seed*, *evaluator*)

**simple_run**(*population_size: int*, *n_generations: int*, *cxpb: float = 0.5*, *mutation_prob: float = 0.5*, *n_elites: int = 10*, *tournsize: int = 3*, *indpb: float = 0.05*)

Method to run the genetic algorithm. This uses the deap eaSimple method. It cannot be used to track what happens in each generation.

**Parameters**

- **population_size** (*int*) – size of the population
- **n_generations** (*int*) – number of generations
- **cxpb** (*float*) – crossover probability
- **mutation_prob** (*float*) – mutation probability
- **n_elites** (*int*) – number of elites
- **tournsize** (*int*) – size of the tournament
- **indpb** (*float*) – probability of a gene to be mutated

**Returns**

- **population** (*list*) – final population
- **logbook** (*~deap.tools.Logbook*) – logbook
- **hof** (*~deap.tools.HallOfFame*) – hall of fame

**run**(*population_size: int*, *n_generations: int*, *cxpb: float = 0.5*, *mutation_prob: float = 0.5*, *n_elites: int = 10*, *tournsize: int = 3*, *indpb: float = 0.05*, *checkpoint: str = None*) → object

Method to run the genetic algorithm. This uses the custom_ea_simple method. It allows to track what happens in each generation.

**Parameters**

- **population_size** (*int*) – size of the population
- **n_generations** (*int*) – number of generations
- **cxpb** (*float*) – crossover probability
- **mutation_prob** (*float*) – mutation probability
- **n_elites** (*int*) – number of elites
- **tournsize** (*int*) – size of the tournament

- **indpb** (*float*) – probability of a gene to be mutated
- **checkpoint** (*str*) – path to the checkpoint file

**Returns**

- **population** (*list*) – final population
- **logbook** (*~deap.tools.Logbook*) – logbook
- **hof** (*~deap.tools.HallOfFame*) – hall of fame

**custom_ea_simple**(*population: list*, *toolbox: deap.base.Toolbox*, *cxpb: float = 0.5*, *mutpb: float = 0.5*, *start_gen: int = 0*, *ngen: int = 4*, *checkpoint_path: str = None*, *stats: deap.tools.Statistics = None*, *halloffame: deap.tools.HallOfFame = None*, *verbose: bool = True*, *checkpoint_flag: bool = True*)

This algorithm reproduces the simplest evolutionary algorithm as presented in chapter 7 of [Back2000].

The code is close to the ~deap.algorithms.eaSimple method, but it has been modified to track the progress of the optimization and to save the population and the logbook in each generation. More info can be found on deap documentation

**Parameters**

- **population** (*list*) – A list of individuals.
- **toolbox** (*Toolbox*) – A *toolbox* that contains the evolution operators.
- **cxpb** (*float*) – The probability of mating two individuals.
- **mutpb** (*float*) – The probability of mutating an individual.
- **start_gen** (*int*) – The starting generation number. Used in case of checkpoint.
- **ngen** (*int*) – The number of generations.
- **checkpoint_path** (*str*) – The path to the checkpoint file.
- **stats** (*Statistics*) – A *~deap.tools.Statistics* object that is updated inplace, optional.
- **halloffame** (*HallOfFame*) – A *~deap.tools.HallOfFame* object that contains the best individuals, optional.
- **verbose** (*bool*) – Whether or not to log the statistics.
- **checkpoint_flag** (*bool*) – Whether or not to save the checkpoint.

**Returns**

- **population** (*list*) – The final population.
- **logbook** (*~deap.tools.Logbook*) – A logbook containing the statistics of the evolution.
- **halloffame** (*~deap.tools.HallOfFame*) – A hall of fame object that contains the best individuals.

### References

**population_2_df**()

Method to convert the population to a pandas dataframe

**Returns**
**df** – dataframe with the population

**Return type**
pandas dataframe

## mloptimizer.genetic.individual

### Module Contents

### Classes

| |
|---|
| *IndividualUtils* |

**class** mloptimizer.genetic.individual.**IndividualUtils**(*hyperparam_space:* mloptimizer.hyperparams.HyperparameterSpace = *None*, *estimator_class=None*, *mlopt_seed=None*)

**get_clf**(*individual*)

**individual2dict**(*individual*)

Method to convert an individual to a dictionary of hyperparams

**Parameters**
**individual** (*individual*) – individual to convert

**Returns**
**individual_dict** – dictionary of hyperparams

**Return type**
dict

### Package Contents

### Classes

| |
|---|
| *DeapOptimizer* |
| *GeneticAlgorithmRunner* |
| *IndividualUtils* |

class mloptimizer.genetic.**DeapOptimizer**(*hyperparam_space:* mloptimizer.hyperparams.HyperparameterSpace = *None*, *use_parallel=False*, *seed=None*)

**init_individual**(*pcls*)

Method to create an individual

**Parameters**
**pcls** (*class*) – class of the individual

**Returns**
**ind** – individual

**Return type**
individual

**individual2dict**(*individual*)

Method to convert an individual to a dictionary of hyperparams

**Parameters**
**individual** (*individual*) – individual to convert

**Returns**
**individual_dict** – dictionary of hyperparams

**Return type**
dict

**setup**()

Method to set the parameters for the optimization.

class mloptimizer.genetic.**GeneticAlgorithmRunner**(*deap_optimizer:* mloptimizer.genetic.DeapOptimizer, *tracker:* mloptimizer.aux.Tracker, *seed*, *evaluator*)

**simple_run**(*population_size: int*, *n_generations: int*, *cxpb: float = 0.5*, *mutation_prob: float = 0.5*, *n_elites: int = 10*, *tournsize: int = 3*, *indpb: float = 0.05*)

Method to run the genetic algorithm. This uses the deap eaSimple method. It cannot be used to track what happens in each generation.

**Parameters**

- **population_size** (*int*) – size of the population
- **n_generations** (*int*) – number of generations
- **cxpb** (*float*) – crossover probability
- **mutation_prob** (*float*) – mutation probability
- **n_elites** (*int*) – number of elites
- **tournsize** (*int*) – size of the tournament
- **indpb** (*float*) – probability of a gene to be mutated

**Returns**

- **population** (*list*) – final population
- **logbook** (*~deap.tools.Logbook*) – logbook
- **hof** (*~deap.tools.HallOfFame*) – hall of fame

**run**(*population_size: int*, *n_generations: int*, *cxpb: float = 0.5*, *mutation_prob: float = 0.5*, *n_elites: int = 10*, *tournsize: int = 3*, *indpb: float = 0.05*, *checkpoint: str = None*) → object

Method to run the genetic algorithm. This uses the custom_ea_simple method. It allows to track what happens in each generation.

**Parameters**

- **population_size** (*int*) – size of the population
- **n_generations** (*int*) – number of generations
- **cxpb** (*float*) – crossover probability
- **mutation_prob** (*float*) – mutation probability
- **n_elites** (*int*) – number of elites
- **tournsize** (*int*) – size of the tournament
- **indpb** (*float*) – probability of a gene to be mutated
- **checkpoint** (*str*) – path to the checkpoint file

**Returns**

- **population** (*list*) – final population
- **logbook** (*~deap.tools.Logbook*) – logbook
- **hof** (*~deap.tools.HallOfFame*) – hall of fame

**custom_ea_simple**(*population: list*, *toolbox: deap.base.Toolbox*, *cxpb: float = 0.5*, *mutpb: float = 0.5*, *start_gen: int = 0*, *ngen: int = 4*, *checkpoint_path: str = None*, *stats: deap.tools.Statistics = None*, *halloffame: deap.tools.HallOfFame = None*, *verbose: bool = True*, *checkpoint_flag: bool = True*)

This algorithm reproduces the simplest evolutionary algorithm as presented in chapter 7 of [Back2000].

The code is close to the ~deap.algorithms.eaSimple method, but it has been modified to track the progress of the optimization and to save the population and the logbook in each generation. More info can be found on deap documentation

**Parameters**

- **population** (*list*) – A list of individuals.
- **toolbox** (*Toolbox*) – A *toolbox* that contains the evolution operators.
- **cxpb** (*float*) – The probability of mating two individuals.
- **mutpb** (*float*) – The probability of mutating an individual.
- **start_gen** (*int*) – The starting generation number. Used in case of checkpoint.
- **ngen** (*int*) – The number of generations.
- **checkpoint_path** (*str*) – The path to the checkpoint file.
- **stats** (*Statistics*) – A *~deap.tools.Statistics* object that is updated inplace, optional.
- **halloffame** (*HallOfFame*) – A *~deap.tools.HallOfFame* object that contains the best individuals, optional.
- **verbose** (*bool*) – Whether or not to log the statistics.
- **checkpoint_flag** (*bool*) – Whether or not to save the checkpoint.

**Returns**

- **population** (*list*) – The final population.
- **logbook** (*~deap.tools.Logbook*) – A logbook containing the statistics of the evolution.
- **halloffame** (*~deap.tools.HallOfFame*) – A hall of fame object that contains the best individuals.

#### References

**population_2_df**()

Method to convert the population to a pandas dataframe

**Returns**
**df** – dataframe with the population

**Return type**
pandas dataframe

class mloptimizer.genetic.**IndividualUtils**(*hyperparam_space: mloptimizer.hyperparams.HyperparameterSpace = None*, *estimator_class=None*, *mlopt_seed=None*)

**get_clf**(*individual*)

**individual2dict**(*individual*)

Method to convert an individual to a dictionary of hyperparams

**Parameters**
**individual** (*individual*) – individual to convert

**Returns**
**individual_dict** – dictionary of hyperparams

**Return type**
dict

## mloptimizer.hyperparams

### Submodules

## mloptimizer.hyperparams.hyperparam

### Module Contents

### Classes

| | |
|---|---|
| *Hyperparam* | Class to define a hyperparam to optimize. It defines the name, min value, max value and type. |

class mloptimizer.hyperparams.hyperparam.**Hyperparam**(*name: str*, *min_value: int*, *max_value: int*, *hyperparam_type: str*, *scale: int = 100*, *values_str: list = None*)

Bases: `object`

Class to define a hyperparam to optimize. It defines the name, min value, max value and type. This is used to control the precision of the hyperparam and avoid multiple evaluations with close values of the hyperparam due to decimal positions.

**name**

Name of the hyperparam. It will be used as key in a dictionary

**Type**
str

**min_value**

Minimum value of the hyperparam

**Type**
int

**max_value**

Maximum value of the hyperparam

**Type**
int

**hyperparam_type**

Type of the hyperparam ('int', 'float', 'nexp', 'x10')

**Type**
str

**scale**

Optional param in case the type=float

**Type**
int, optional (default=100)

**values_str**

List of string with possible values (TODO)

**Type**
list, optional (default=[])

**correct**(*value: int*)

**Returns the real value of the hyperparam in case some mutation could surpass the limits.**

1) Verifies the input is int
2) Enforce min and max value
3) Apply the type of value

**Parameters**
**value** (*int*) – Value to correct

**Returns**
**ret** – Corrected value

**Return type**
int, float

### mloptimizer.hyperparams.hyperspace

## Module Contents

## Classes

| | |
|---|---|
| *HyperparameterSpace* | This class represents the hyperparameter space for a scikit-learn classifier. It contains the fixed hyperparameters |

**class** mloptimizer.hyperparams.hyperspace.**HyperparameterSpace**(*fixed_hyperparams: dict*, *evolvable_hyperparams: dict*)

This class represents the hyperparameter space for a scikit-learn classifier. It contains the fixed hyperparameters and the evolvable hyperparameters. The fixed hyperparameters are just a dictionary with the hyperparameters that are not going to be optimized and their value. The evolvable hyperparameters are a dictionary with the hyperparameters that are going to be optimized. The keys are the hyperparameter names and the values are instances of the *Hyperparam* class.

**fixed_hyperparams**

Dictionary with the fixed hyperparameters

**Type**
dict

**evolvable_hyperparams**

Dictionary with the evolvable hyperparameters of *Hyperparam* instances

**Type**
dict

**default_hyperparameter_spaces_json**

**classmethod from_json**(*file_path*)

This method creates a *HyperparameterSpace* object from a JSON file.

**Parameters**
**file_path** (*str*) – Path to the JSON file

**Return type**
*HyperparameterSpace*

**Raises**

- **FileNotFoundError** – If the file does not exist
- **json.JSONDecodeError** – If the file is not a valid JSON file

**to_json**(*file_path*, *overwrite=False*)

This method saves the hyperparameter space as a JSON file.

**Parameters**

- **file_path** (*str*) – Path to the JSON file
- **overwrite** (*bool, optional (default=False)*) – If True, the file will be overwritten if it exists. If False, a FileExistsError will be raised if the file exists

**Raises**

- **ValueError** – If the file path is None
- **FileExistsError** – If the file exists and overwrite is False

**static get_default_hyperparameter_space**(*estimator_class*)

This method returns a dictionary with the default hyperparameters for the scikit-learn classifier. It reads the default_hyperparameter_spaces.json file and returns the hyperparameters for the classifier

**Parameters**
**estimator_class** (*class*) – The scikit-learn classifier class

**Returns**
The hyperparameter space for the classifier

**Return type**
*HyperparameterSpace*

## Package Contents

## Classes

| | |
|---|---|
| *Hyperparam* | Class to define a hyperparam to optimize. It defines the name, min value, max value and type. |
| *HyperparameterSpace* | This class represents the hyperparameter space for a scikit-learn classifier. It contains the fixed hyperparameters |

**class** mloptimizer.hyperparams.**Hyperparam**(*name: str*, *min_value: int*, *max_value: int*, *hyperparam_type: str*, *scale: int = 100*, *values_str: list = None*)

Bases: object

Class to define a hyperparam to optimize. It defines the name, min value, max value and type. This is used to control the precision of the hyperparam and avoid multiple evaluations with close values of the hyperparam due to decimal positions.

**name**

Name of the hyperparam. It will be used as key in a dictionary

**Type**
str

**min_value**

Minimum value of the hyperparam

**Type**
int

**max_value**

Maximum value of the hyperparam

**Type**
int

**hyperparam_type**

Type of the hyperparam ('int', 'float', 'nexp', 'x10')

**Type**
str

`scale`

Optional param in case the type=float

**Type**
int, optional (default=100)

`values_str`

List of string with possible values (TODO)

**Type**
list, optional (default=[])

`correct`(*value: int*)

**Returns the real value of the hyperparam in case some mutation could surpass the limits.**

1) Verifies the input is int
2) Enforce min and max value
3) Apply the type of value

**Parameters**
**value** (*int*) – Value to correct

**Returns**
**ret** – Corrected value

**Return type**
int, float

`class mloptimizer.hyperparams.HyperparameterSpace`(*fixed_hyperparams: dict*, *evolvable_hyperparams: dict*)

This class represents the hyperparameter space for a scikit-learn classifier. It contains the fixed hyperparameters and the evolvable hyperparameters. The fixed hyperparameters are just a dictionary with the hyperparameters that are not going to be optimized and their value. The evolvable hyperparameters are a dictionary with the hyperparameters that are going to be optimized. The keys are the hyperparameter names and the values are instances of the *Hyperparam* class.

`fixed_hyperparams`

Dictionary with the fixed hyperparameters

**Type**
dict

`evolvable_hyperparams`

Dictionary with the evolvable hyperparameters of *Hyperparam* instances

**Type**
dict

`default_hyperparameter_spaces_json`

`classmethod from_json`(*file_path*)

This method creates a *HyperparameterSpace* object from a JSON file.

**Parameters**
**file_path** (*str*) – Path to the JSON file

**Return type**
*HyperparameterSpace*

**Raises**

- **FileNotFoundError** – If the file does not exist
- **json.JSONDecodeError** – If the file is not a valid JSON file

**to_json**(*file_path*, *overwrite=False*)

This method saves the hyperparameter space as a JSON file.

**Parameters**

- **file_path** (*str*) – Path to the JSON file
- **overwrite** (*bool, optional (default=False)*) – If True, the file will be overwritten if it exists. If False, a FileExistsError will be raised if the file exists

**Raises**

- **ValueError** – If the file path is None
- **FileExistsError** – If the file exists and overwrite is False

**static get_default_hyperparameter_space**(*estimator_class*)

This method returns a dictionary with the default hyperparameters for the scikit-learn classifier. It reads the default_hyperparameter_spaces.json file and returns the hyperparameters for the classifier

**Parameters**
**estimator_class** (*class*) – The scikit-learn classifier class

**Returns**
The hyperparameter space for the classifier

**Return type**
*HyperparameterSpace*

CHAPTER

NINE

# DEVELOPMENT

This page is here, to provide some content for the site structure.

CHAPTER

TEN

# CHANGELOG

This document provides a comprehensive list of changes for each version of the mloptimizer library.

## 10.1 Version 0.6.0 (2024-01-24)

### 10.1.1 "Initial" Release

- Launched the first version of the mloptimizer library.
- Features include optimization for decision trees, random forests, and gradient boosting classifiers.
- Basic documentation and example scripts provided.

Please refer to the [GitHub repository](https://github.com/Caparrini/mloptimizer) for a detailed list of changes.

## 10.2 Template - Version X.X.X (YYYY-MM-DD)

### 10.2.1 Features

- Feature 1

### 10.2.2 Bug Fixes

- Bug fix 1

### 10.2.3 Improvements

- Improvement 1

CHAPTER

ELEVEN

# INDICES AND TABLES

- genindex
- modindex
- search

# BIBLIOGRAPHY

[Back2000] Back, Fogel and Michalewicz, “Evolutionary Computation 1 : Basic Algorithms and Operators”, 2000.
[Back2000] Back, Fogel and Michalewicz, “Evolutionary Computation 1 : Basic Algorithms and Operators”, 2000.

# PYTHON MODULE INDEX

## m

# INDEX

## F



## G



## H



## I



## K

## O



## P



## R



## S



## T



## U

## V



## W