sklearn genetic opt

Release 0.9.0

Rodrigo Arenas Gómez

Jan 20, 2023

USER GUIDE / TUTORIALS:

scikit-learn models hyperparameters tuning and feature selection, using evolutionary algorithms.

This is meant to be an alternative to popular methods inside scikit-learn such as Grid Search and Randomized Grid Search for hyperparameters tuning, and from RFE, Select From Model for feature selection.

Sklearn-genetic-opt uses evolutionary algorithms from the deap package to choose a set of hyperparameters that optimizes (max or min) the cross-validation scores, it can be used for both regression and classification problems.

CHAPTER

ONE

INSTALLATION:

Install sklearn-genetic-opt

It's advised to install sklearn-genetic using a virtual env, to install a light version, inside the env use:

```
pip install sklearn-genetic-opt
```

sklearn-genetic-opt requires:

- Python (>= 3.7)
- scikit-learn (>= 0.21.3)
- NumPy (>= 1.14.5)
- DEAP (>= 1.3.1)
- tqdm (>= 4.61.1)

Extra requirements:

These requirements are necessary to use *plots*, `MLflowConfig` and *TensorBoard* correspondingly.

- Seaborn (>= 0.9.0)
- MLflow (>= 1.17.0)
- Tensorflow (>= 2.0.0)

This command will install all the extra requirements, except for Tensorflow, as it is usually advised to look further which distribution works better for you:

```
pip install sklearn-genetic-opt[all]
```

1.1 How to Use Sklearn-genetic-opt

1.1.1 Introduction

Sklearn-genetic-opt uses evolutionary algorithms to fine-tune scikit-learn machine learning algorithms and perform feature selection. It is designed to accept a scikit-learn regression or classification model (or a pipeline containing on of those).

The idea behind this package is to define the set of hyperparameters we want to tune and what are their lower and uppers bounds on the values they can take. It is possible to define different optimization algorithms, callbacks and build-in parameters to control how the optimization is taken. To get started, we'll use only the most basic features and options.

The optimization is made by evolutionary algorithms with the help of the deap package. It works by defining the set of hyperparameters to tune, it starts with a randomly sampled set of options (population). Then by using evolutionary operators as the mating, mutation, selection and evaluation, it generates new candidates looking to improve the cross-validation score in each generation. It'll continue with this process until a number of generations is reached or until a callback criterion is met.

1.1.2 Fine-tuning Example

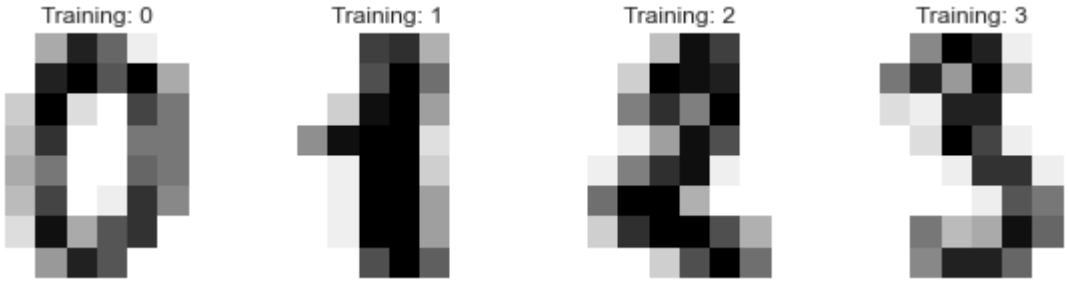
First let's import some dataset and other scikit-learn standard modules, we'll use the digits dataset. This is a classification problem, we'll fine-tune a Random Forest Classifier for this task.

```
import matplotlib.pyplot as plt
from sklearn_genetic import GASearchCV
from sklearn_genetic.space import Categorical, Integer, Continuous
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import load_digits
from sklearn.metrics import accuracy_score
```

Let's first read the data, split it in our training and test set and visualize some of the data points:

```
data = load_digits()
n_samples = len(data.images)
X = data.images.reshape((n_samples, -1))
y = data['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_
↪state=42)

_, axes = plt.subplots(nrows=1, ncols=4, figsize=(10, 3))
for ax, image, label in zip(axes, data.images, data.target):
    ax.set_axis_off()
    ax.imshow(image, cmap=plt.cm.gray_r, interpolation='nearest')
    ax.set_title('Training: %i' % label)
```

We should see something like this:



Now, we must define our param_grid, similar to scikit-learn, which is a dictionary with the model's hyperparameters. The main difference with for example sckit-learn's GridSearchCv, is that we don't pre-define the values to use in the search, but rather, the boundaries of each parameter.

So if we have a parameter named *'n_estimators'* we'll only tell to sckit-learn-genetic-opt, that is an integer value, and that we want to set a lower boundary of 100 and an upper boundary of 500, so the optimizer will set a value in this range. We must do this with all the hyperparameters we want to tune, like this:

```
param_grid = {'min_weight_fraction_leaf': Continuous(0.01, 0.5, distribution='log-uniform
↪'),
              'bootstrap': Categorical([True, False]),
              'max_depth': Integer(2, 30),
              'max_leaf_nodes': Integer(2, 35),
              'n_estimators': Integer(100, 300)}
```

Notice that in the case of *'boostrap'*, as it is a categorical variable, we do must define all its possible values. As well, in the 'min_weight_fraction_leaf', we used an additional parameter named distribution, this is useful to tell the optimizer from which data distribution it can sample some random values during the optimization.

Now, we are ready to set the GASearchCV, its the object that will allow us to run the fitting process using evolutionary algorithms It has several options that we can use, for this first example, we'll keep it very simple:

```
# The base classifier to tune
clf = RandomForestClassifier()

# Our cross-validation strategy (it could be just an int)
cv = StratifiedKFold(n_splits=3, shuffle=True)

# The main class from sklearn-genetic-opt
evolved_estimator = GASearchCV(estimator=clf,
                               cv=cv,
                               scoring='accuracy',
                               param_grid=param_grid,
                               n_jobs=-1,
                               verbose=True)
```

So now the setup is ready, note that are other parameters that can be specified in GASearchCV, the ones we used, are equivalents to the meaning in scikit-learn, besides the one already explained, is worth mentioning that the "metric" is going to be used as the optimization variable, so the algorithm will try to find the set of parameters that maximizes this metric.

We are ready to run the optimization routine:

```
 # Train and optimize the estimator
evolved_estimator.fit(X_train, y_train)
```

During the training process, you should see a log like this:

```
1 evolved_estimator.fit(X_train,y_train)
```

gen	nevals	fitness	fitness_std	fitness_max	fitness_min
0	10	0.799258	0.123352	0.920143	0.541734
1	14	0.869397	0.0468278	0.909021	0.799679
2	20	0.902797	0.0214169	0.922088	0.862663
3	17	0.923556	0.00421409	0.929485	0.918343
4	19	0.923001	0.00551673	0.929485	0.910946
5	17	0.926705	0.00432808	0.929485	0.918343
6	19	0.925591	0.00486938	0.929485	0.918343
7	19	0.926145	0.00516765	0.929485	0.916511
8	19	0.923742	0.00628633	0.929485	0.914618
9	19	0.921704	0.00592012	0.929485	0.910935
10	16	0.925788	0.00405725	0.929485	0.918343
11	19	0.922245	0.0077258	0.929485	0.905328
12	19	0.918746	0.00703819	0.931357	0.909114
13	19	0.924705	0.00123214	0.927664	0.92396
14	20	0.924309	0.00262313	0.927664	0.920174
15	17	0.925981	0.00410335	0.931368	0.918405
16	17	0.928951	0.00462616	0.938806	0.920226
17	17	0.935635	0.00343967	0.938827	0.929495
18	17	0.938045	0.00170187	0.940648	0.935051
19	20	0.939145	0.00258897	0.946182	0.936923
20	18	0.941001	0.0048046	0.946182	0.931409
21	17	0.94137	0.00439765	0.948045	0.936872
22	18	0.941565	0.00663459	0.949897	0.929526
23	18	0.94618	0.0046269	0.949897	0.938775
24	19	0.945076	0.00525547	0.949897	0.936954
25	17	0.940621	0.00974303	0.949897	0.927623
26	16	0.944328	0.00563166	0.949897	0.936923
27	16	0.945627	0.00538466	0.949897	0.936923
28	17	0.944888	0.00615387	0.949897	0.936913
29	18	0.94378	0.00682917	0.949897	0.931419
30	18	0.943032	0.008666	0.949897	0.927612
31	18	0.945633	0.00658267	0.949897	0.933209
32	17	0.943215	0.00704567	0.949897	0.931337
33	14	0.943595	0.0062123	0.949897	0.933209
34	18	0.942837	0.00591748	0.949897	0.93504
35	20	0.94378	0.00579118	0.949897	0.935133

This log, shows us the metrics obtained in each iteration (generation), this is what each entry means:

- **gen:** The number of the generation
- **nevals:** How many hyperparameters were fitted in this generation
- **fitness:** The average score metric in the cross-validation (validation set). In this case, the average accuracy across the folds of all the hyperparameters sets.
- **fitness_std:** The standard deviation of the cross-validations accuracy.
- **fitness_max:** The maximum individual score of all the models in this generation.
- **fitness_min:** The minimum individual score of all the models in this generation.

After fitting the model, we have some extra methods to use the model right away. It will use by default the best set of hyperparameters it found, based on the cross-validation score:

```
# Best parameters found
print(evolved_estimator.best_params_)
# Use the model fitted with the best parameters
y_predict_ga = evolved_estimator.predict(X_test)
print(accuracy_score(y_test, y_predict_ga))
```

In this case, we got an accuracy score in the test set of 0.93

```
y_predicy_ga = evolved_estimator.predict(X_test)
accuracy_score(y_test,y_predicy_ga)
```

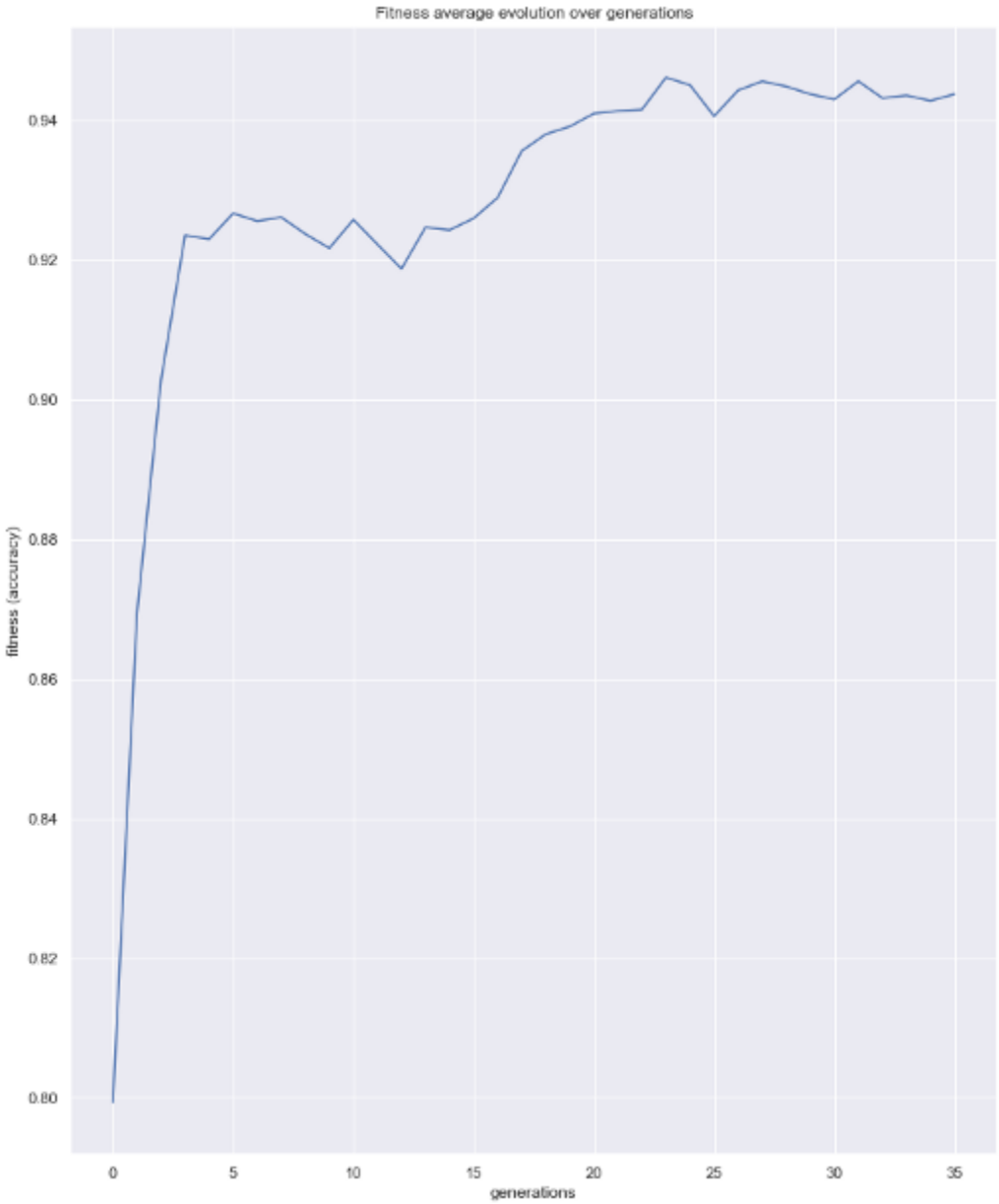
```
0.9340222575516693
```

```
evolved_estimator.best_params
```

```
{'min_weight_fraction_leaf': 0.014725004803419667,
 'bootstrap': True,
 'max_depth': 25,
 'max_leaf_nodes': 29,
 'n_estimators': 259}
```

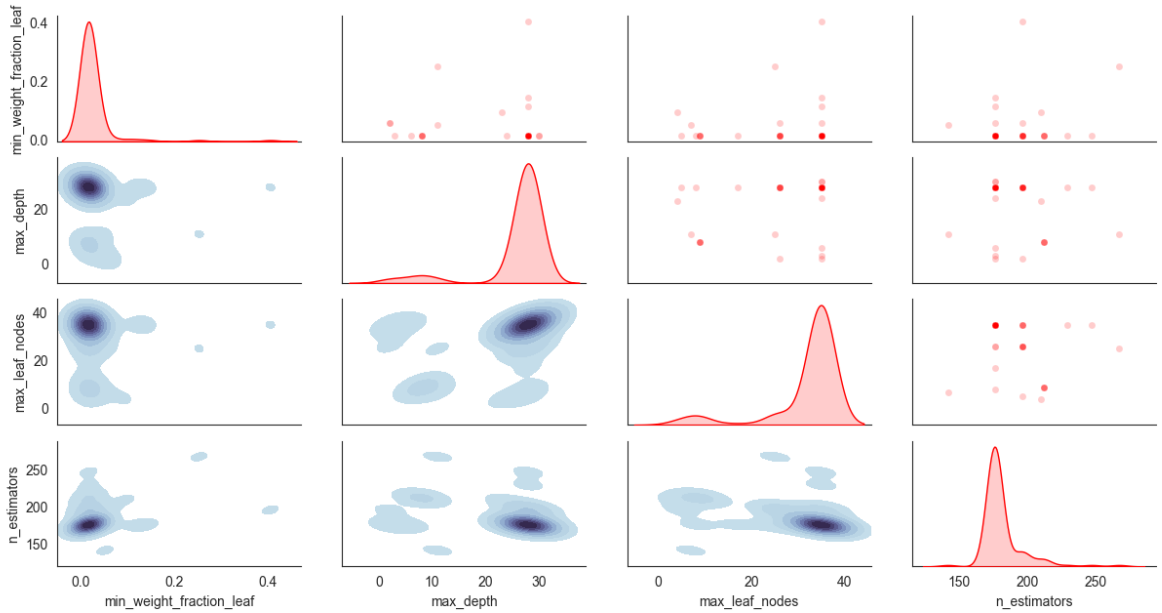
Now, let's use a couple more functions available in the package. The first one will help us to see the evolution of our metric over the generations

```
from sklearn_genetic.plots import plot_fitness_evolution
plot_fitness_evolution(evolved_estimator)
plt.show()
```



At last, we can check the property called `evolved_estimator.logbook`, this is a DEAP's logbook which stores all the results of every individual fitted model. sklearn-genetic-opt comes with a plot function to analyze this log:

```
from sklearn_genetic.plots import plot_search_space
plot_search_space(evolved_estimator, features=['min_weight_fraction_leaf', 'max_depth',
↪'max_leaf_nodes', 'n_estimators'])
plt.show()
```



What this plot shows us, is the distribution of the sampled values for each hyperparameter. We can see for example in the *'min_weight_fraction_leaf'* that the algorithm mostly sampled values below 0.15. You can also check every single combination of variables and the contour plot that represents the sampled values.

1.1.3 Feature Selection Example

For this example, we are going to use the well-known Iris dataset, it's a classification problem with four features. We are also going to simulate some random noise to represent non-important features:

```
import matplotlib.pyplot as plt
from sklearn_genetic import GAFeatureSelectionCV
from sklearn_genetic.plots import plot_fitness_evolution
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.svm import SVC
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score
import numpy as np

data = load_iris()
X, y = data["data"], data["target"]

noise = np.random.uniform(0, 10, size=(X.shape[0], 10))

X = np.hstack((X, noise))

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=0)
```

This should give us 10 extra noisy features with our train and test set.

Now we can create the GAFeatureSelectionCV object, it's very similar to the GASearchCV and they share most of the parameters, the main difference is GAFeatureSelectionCV doesn't run hyperparameters optimization thus the param_grid parameter it's not available, and the estimator should be defined with its hyperparameters.

The way the feature selection is performed is by creating models with a subsample of features and evaluate its cv-score,

the way the subsets are created is by using the available evolutionary algorithms. It also tries to minimize the number of selected features, so it's a multi-objective optimization.

Let's create the feature selection object, the estimator we're going to use is a SVM:

```
clf = SVC(gamma='auto')

evolved_estimator = GAFeatureSelectionCV(
    estimator=clf,
    cv=3,
    scoring="accuracy",
    population_size=30,
    generations=20,
    n_jobs=-1,
    verbose=True,
    keep_top_k=2,
    elitism=True,
)
```

We are ready to run the optimization routine:

```
# Train and select the features
evolved_estimator.fit(X_train, y_train)
```

During the training, the same log format is displayed as before:

gen	nevals	fitness	fitness_std	fitness_max	fitness_mir
0	30	0.558444	0.155441	0.893333	0.253333
1	54	0.659333	0.132948	0.893333	0.333333
2	54	0.742667	0.0867111	0.893333	0.586667
3	55	0.805778	0.0740117	0.893333	0.653333
4	52	0.873333	0.0435125	0.906667	0.746667
5	53	0.896222	0.00659592	0.913333	0.893333
6	55	0.901111	0.0131186	0.953333	0.893333
7	54	0.911778	0.0206332	0.953333	0.893333
8	50	0.926444	0.0210455	0.953333	0.893333
9	51	0.941333	0.020177	0.966667	0.913333
10	49	0.955556	0.00978787	0.966667	0.913333
11	55	0.959111	0.00660714	0.966667	0.953333
12	57	0.965333	0.004	0.966667	0.953333
13	55	0.966444	0.00271257	0.973333	0.953333
14	58	0.966667	6.66134e-16	0.966667	0.966667
15	53	0.966889	0.0011967	0.973333	0.966667
16	56	0.967556	0.00226623	0.973333	0.966667
17	53	0.969556	0.00330357	0.973333	0.966667
18	51	0.971111	0.0031427	0.973333	0.966667
19	58	0.972889	0.00166296	0.973333	0.966667
20	54	0.973333	3.33067e-16	0.973333	0.973333

After fitting the model, we have some extra methods to use the model right away. It will use by default the best set of features it found, remember as the algorithm used only a subset, you have to select them from the `X_test array`, this is done like this:

```
features = evolved_estimator.best_features_

# Predict only with the subset of selected features
y_predict_ga = evolved_estimator.predict(X_test[:, features])
accuracy = accuracy_score(y_test, y_predict_ga)
```

```
print(evolved_estimator.best_features_)
print("accuracy score: ", "{:.2f}".format(accuracy))
```

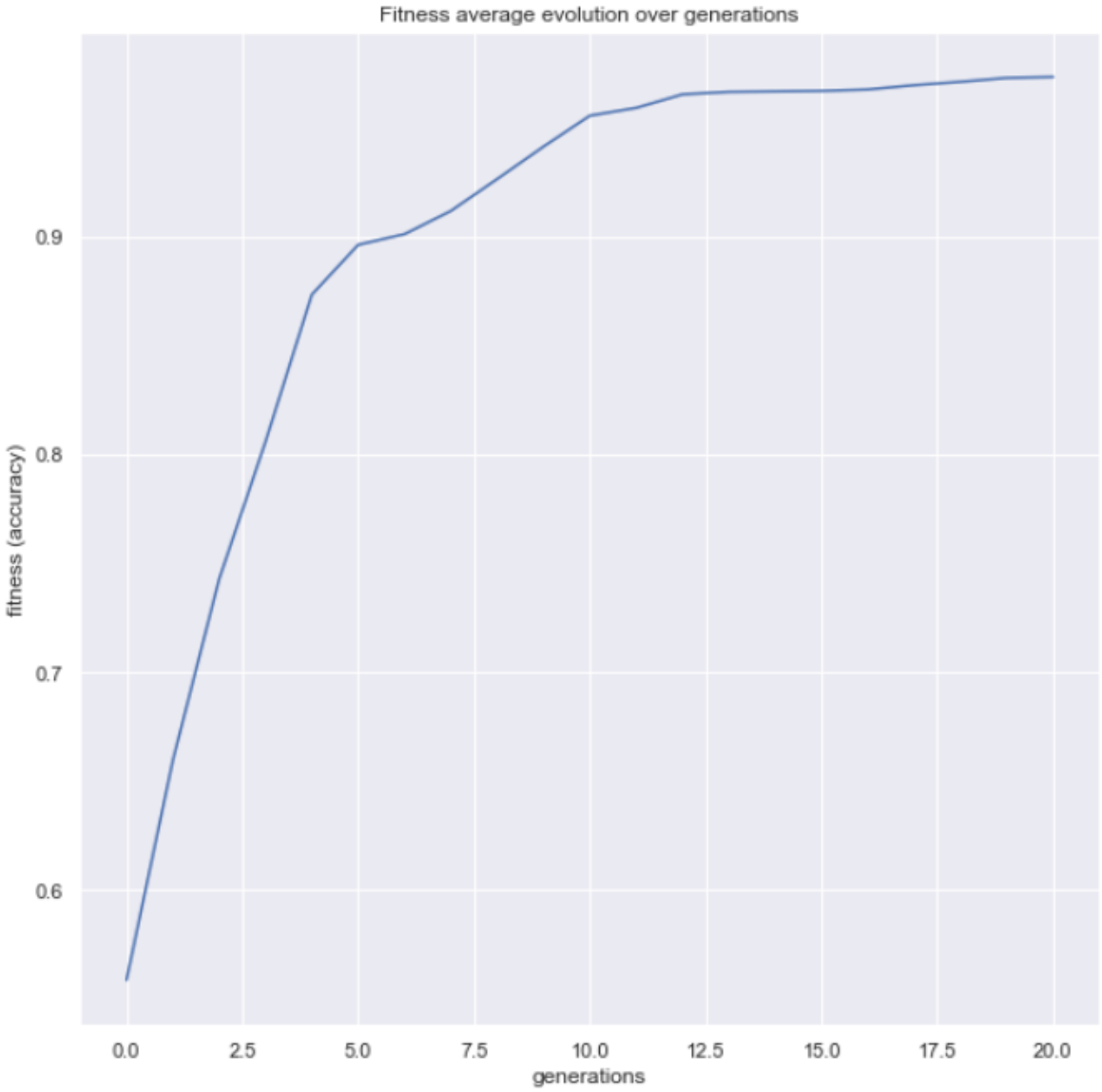
```
[ True  True  True  True False False False False False False False False
 False False]
accuracy score:  0.98
```

In this case, we got an accuracy score in the test set of 0.98.

Notice that the `best_features_` is a vector of bool values, each position represents the index of the feature (column) and the value indicates if that features was selected (True) or not (False) by the algorithm. In this example, the algorithm, discarded all the noisy random variables we created and selected the original variables.

We can also plot the fitness evolution:

```
from sklearn_genetic.plots import plot_fitness_evolution
plot_fitness_evolution(evolved_estimator)
plt.show()
```



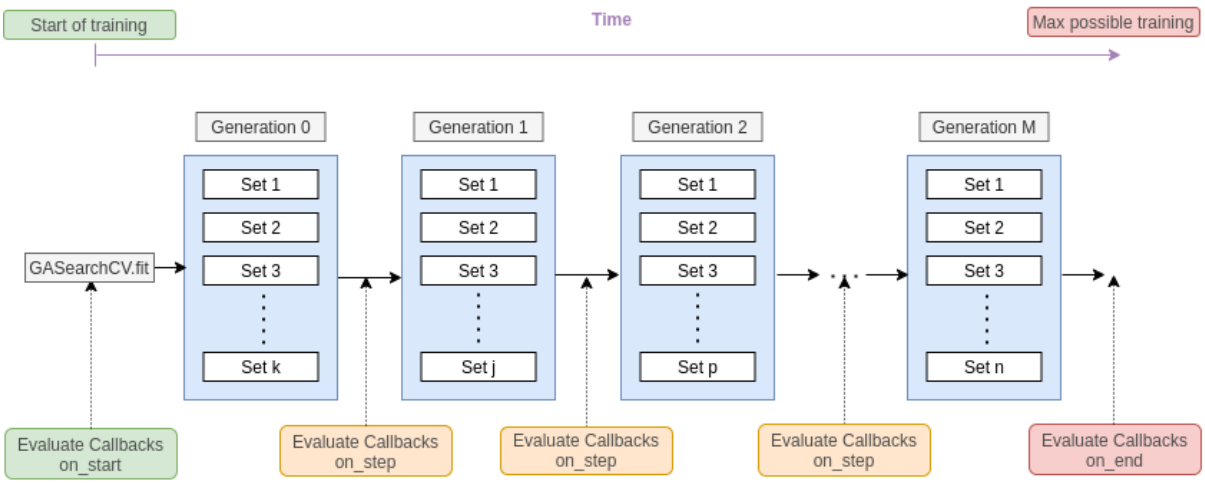
This concludes our introduction to the basic sklearn-genetic-opt usage. Further tutorials will cover the GASearchCV and GAFeatureSelectionCV parameters, callbacks, different optimization algorithms and more advanced use cases.

1.2 Using Callbacks

1.2.1 Introduction

Callbacks can be defined to take actions or decisions over the optimization process while it is still running. Common callbacks include different rules to stop the algorithm or log artifacts. The callbacks are passed to the `.fit` method of the *GASearchCV* or *GAFeatureSelectionCV* class.

The callbacks are evaluated at the start of the training using the *on_start* method, at the end of each generation fit using *on_step* method and at the end of the training using *on_end*, so it looks like this:



When a stopping callback condition is met, we should see a message like this and the model must stop. It will keep all the information until that training point.

```
gen     nevals  fitness         fitness_std     fitness_max     fitness_min
0       16      0.379572        0.200348        0.770574        0.191189
1       29      0.605258        0.146615        0.735661        0.270989
2       32      0.718256        0.0139822       0.738986        0.695761
3       32      0.725322        0.01201         0.738986        0.695761
4       31      0.736128        0.00537084      0.742311        0.724855
5       31      0.73883         0.00319545      0.742311        0.730673
6       29      0.740441        0.00451965      0.74813         0.729842
7       30      0.740337        0.00517348      0.74813         0.729842
INFO: DeltaThreshold callback met its criteria
INFO: Stopping the algorithm
```

Now let's see how to use them, we'll take the data set and model used in *How to Use Sklearn-genetic-opt*. The available callbacks are:

- ProgressBar
- ConsecutiveStopping
- DeltaThreshold
- TimerStopping
- ThresholdStopping
- TensorBoard
- LogbookSaver

1.2.2 ProgressBar

This callback display a tqdm bar with your training process, the length of the bar is the max number of generations (population_size + 1) that the algorithm would run, each step is a generation.

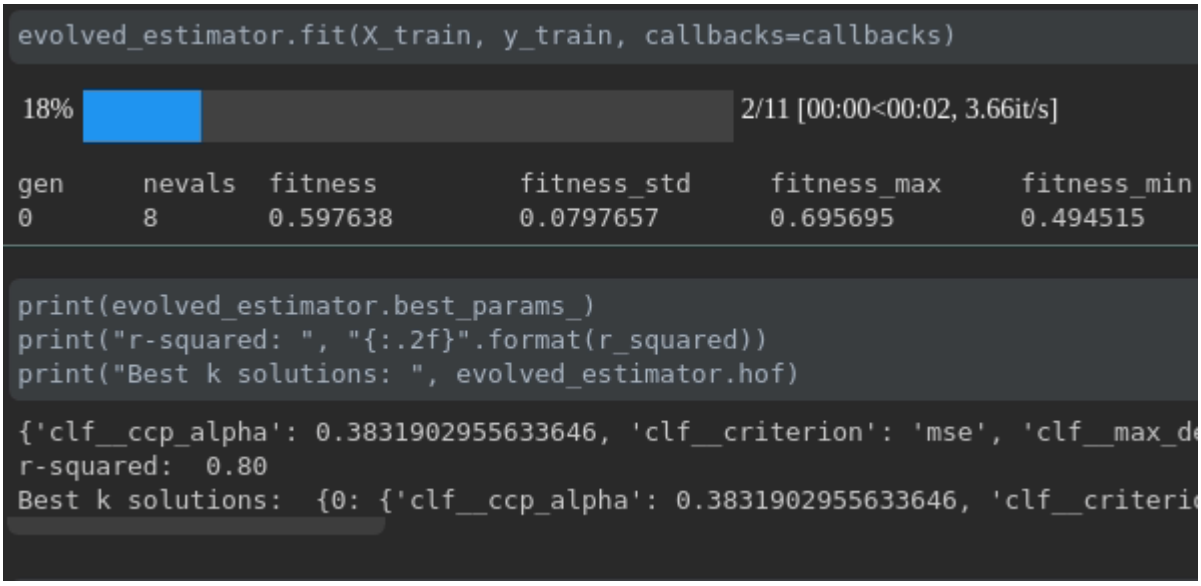
You can pass any `tqdm.auto.tqdm` valid arguments as kwargs or leave it as default. To use this bar set:

```
from sklearn_genetic.callbacks import ProgressBar
callback = ProgressBar()
```

Now we just have to pass it to the estimator during the fitting

```
# Already defined GASearchCV instance
evolved_estimator.fit(X, y, callbacks=callback)
```

During the training it will be displayed like this:



1.2.3 ConsecutiveStopping

This callback stops the optimization if the current metric value is no greater than at least one metric from the last N generations.

It requires us to define the number of generations to compare against the current generation and the name of the metric we want to track.

For example, if we want to stop the optimization after 5 iterations where the current iteration (sixth) fitness value is worst than all the previous ones (5), we define it like this:

```
from sklearn_genetic.callbacks import ConsecutiveStopping
callback = ConsecutiveStopping(generations=5, metric='fitness')
```

Now we just have to pass it to the estimator during the fitting

```
# Already defined GASearchCV instance
evolved_estimator.fit(X, y, callbacks=callback)
```

1.2.4 DeltaThreshold

Stops the optimization if the absolute difference between the maximum and minimum value from the last N generations is less or equals to a threshold.

The threshold gets evaluated after the number of generations specified is reached; the default number is 2 (the current and previous one).

It just requires the threshold, the metric name and the generations, for example using the 'fitness_min' value and comparing the last 5 generations:

```
from sklearn_genetic.callbacks import DeltaThreshold
callback = DeltaThreshold(threshold=0.001, generations=5, metric='fitness_min')

evolved_estimator.fit(X, y, callbacks=callback)
```

1.2.5 TimerStopping

This callback stops the optimization if the difference in seconds between the starting time of the first set of hyperparameters fit, and the current generation time is greater than a time threshold.

Remember that this is checked after each generation fit, so if the first (or any) generation fit takes longer than the threshold, it won't stop the fitting process until is done with the current generation population.

It requires the total_seconds parameters, for example stopping if the time is greater than one minute:

```
from sklearn_genetic.callbacks import TimerStopping
callback = TimerStopping(total_seconds=60)

evolved_estimator.fit(X, y, callbacks=callback)
```

1.2.6 ThresholdStopping

It stops the optimization if the current metric is greater or equals to the defined threshold.

For example, if we want to stop the optimization if the 'fitness_max' is above 0.98:

```
from sklearn_genetic.callbacks import ThresholdStopping
callback = ThresholdStopping(threshold=0.98, metric='fitness_max')

evolved_estimator.fit(X, y, callbacks=callback)
```

1.2.7 TensorBoard

It saves at each iteration the fitness metrics into a log folder that can be read by Tensorboard.

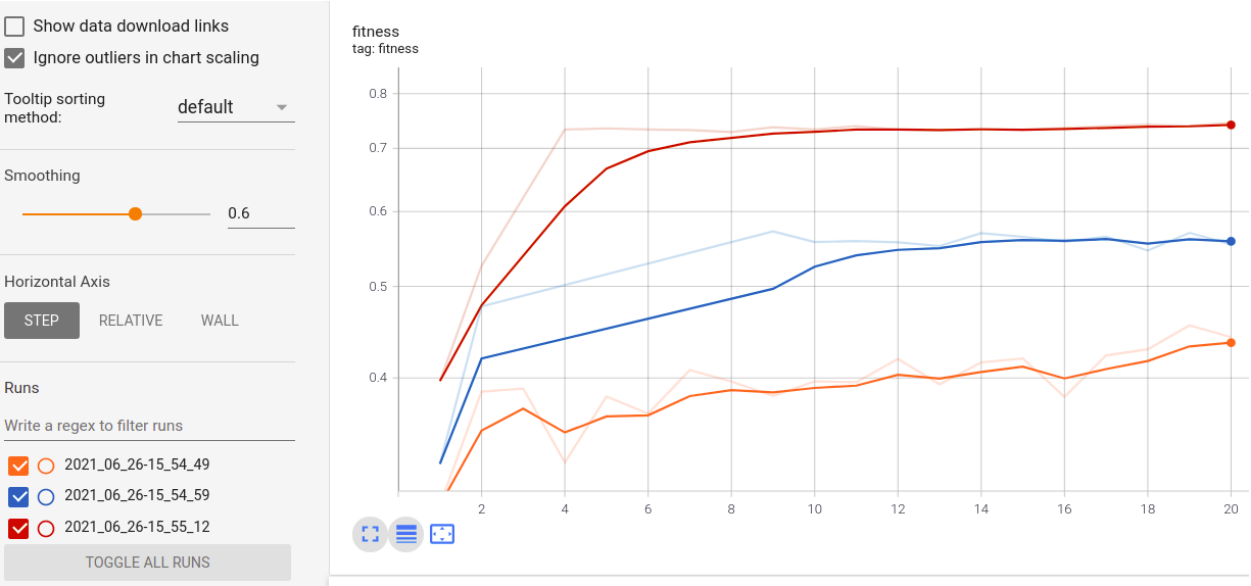
To use this callback you must install tensorflow first, this is not installed within this package due it's usually a sensitive and heavy dependency:

```
pip install tensorflow
```

It only requires defining the folder where you want to log your run, and optionally, a run_id, so your consecutive runs don't mix up. If the run_id is not provided, it will create a subfolder with the current date-time of your run.

```
from sklearn_genetic.callbacks import TensorBoard
callback = TensorBoard(log_dir="./logs")

evolved_estimator.fit(X, y, callbacks=callback)
```

While the model is being trained you can see in real-time the metrics in Tensorboard. If you have run more than one GASearchCV model and use the TensordBoard callback using the same log_dir but different run_id, you can compare the metrics of each run, it looks like this for the fitness in three different runs:



1.2.8 LogbookSaver

It saves at each iteration the Logbook object with all the parameters and the cv-score achieved by those parameters. It uses joblib.dump to save the file.

```
from sklearn_genetic.callbacks import LogbookSaver
callback = LogbookSaver(checkpoint_path="./logbook.pkl")

evolved_estimator.fit(X, y, callbacks=callback)
```

Then the object can be restored:

```
from joblib import load

logbook = load("/.logbook.pkl")
print(logbook)
```

1.2.9 Define Multiple Callbacks

You can also specify more than one callback at the same time. The way to define it is by passing a list of callbacks in the `.fit` method.

Then the estimator is going to check all the conditions in every iteration, if at least one of the stopping callbacks conditions is met, the callback will stop the process:

```
from sklearn_genetic.callbacks import ThresholdStopping, DeltaThreshold
threshold_callback = ThresholdStopping(threshold=0.98, metric='fitness_max')
delta_callback = DeltaThreshold(threshold=0.001, metric='fitness')

callbacks = [threshold_callback, delta_callback]

evolved_estimator.fit(X, y, callbacks=callbacks)
```

1.2.10 Full Example

This example uses a ThresholdStopping and DeltaStopping callback It will stop if the accuracy of the generation is above 0.98 or if the difference between the current generation accuracy and the last generation accuracy is not bigger than 0.001:

```
from sklearn_genetic import GASearchCV
from sklearn_genetic.space import Categorical, Integer, Continuous
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_digits
from sklearn.metrics import accuracy_score
from sklearn_genetic.callbacks import ThresholdStopping, DeltaThreshold


data = load_digits()
label_names = data['target_names']
y = data['target']
X = data['data']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_
↪state=42)

clf = DecisionTreeClassifier()

params_grid = {'min_weight_fraction_leaf': Continuous(0, 0.5),
               'criterion': Categorical(['gini', 'entropy']),
               'max_depth': Integer(2, 20), 'max_leaf_nodes': Integer(2, 30)}

cv = StratifiedKFold(n_splits=3, shuffle=True)

threshold_callback = ThresholdStopping(threshold=0.98, metric='fitness_max')
delta_callback = DeltaThreshold(threshold=0.001, metric='fitness')

callbacks = [threshold_callback, delta_callback]

evolved_estimator = GASearchCV(clf,
```

(continues on next page)

(continued from previous page)

```
                                cv=cv,
                                scoring='accuracy',
                                population_size=16,
                                generations=30,
                                tournament_size=3,
                                elitism=True,
                                crossover_probability=0.9,
                                mutation_probability=0.05,
                                param_grid=params_grid,
                                algorithm='eaMuPlusLambda',
                                n_jobs=-1,
                                verbose=True)

evolved_estimator.fit(X_train, y_train, callbacks=callbacks)
y_predict_ga = evolved_estimator.predict(X_test)
accuracy = accuracy_score(y_test, y_predict_ga)

print(evolved_estimator.best_params_)
print("accuracy score: ", "{:.2f}".format(accuracy))
```

1.3 Custom Callbacks

sklearn-genetic-opt comes with some pre-defined callbacks, but you can make one of your own by defining a callable with certain methods.

1.3.1 Parameters

The callback must be a class with inheritance from the class *BaseCallback* that implements the following methods:

on_start: This is evaluated before starting the generation 0, it should return `None` or `False`. It expects the parameter *estimator*.

on_step: This is called at the end of each generation, the result of them must be a bool, `True` means that the optimization must stop, `False`, means it can continue. It expects the parameters *record*, *logbook* and *estimator*.

on_end: This method is called at the end of the las generation or after a stopping callback meets its criteria. It expects the parameters *logbook* and *estimator*, it should return `None` or `False`.

All of those methods are optional, but at least one should be defined.

1.3.2 Example

In this example, we are going to define a dummy callback that stops the process if there have been more than *N* fitness values bellow a threshold value.

The callback must have three parameters: *record*, *logbook* and *estimator*. Those are a dictionary, a deap's Logbook object and the current *GASearchCV* (or *GAFeatureSelectionCV*) respectively with the current iteration metrics, all the past iterations metrics and all the properties saved in the estimator.

So to check inside the logbook, we could define a function like this:

```
N=4
metric='fitness'
threshold=0.8

def on_step(record, logbook, threshold, estimator=None):
    # Not enough data points
    if len(logbook) <= N:
        return False
    # Get the last N metrics
    stats = logbook.select(metric)[(-N - 1):]

    n_met_condition = [x for x in stats if x < threshold]

    if len(n_met_condition) > N:
        return True

    return False
```

As sklearn-genetic-opt expects all this logic in a single object, we must define a class that will have all these parameters, so we can rewrite it like this:

```
from sklearn_genetic.callbacks.base import BaseCallback

class DummyThreshold(BaseCallback):
    def __init__(self, threshold, N, metric='fitness'):
        self.threshold = threshold
        self.N = N
        self.metric = metric

    def on_step(self, record, logbook, estimator=None):
        # Not enough data points
        if len(logbook) <= self.N:
            return False
        # Get the last N metrics
        stats = logbook.select(self.metric)[(-self.N - 1):]

        n_met_condition = [x for x in stats if x < self.threshold]

        if len(n_met_condition) > self.N:
            return True

        return False
```

Now, let's expend it to add the others method, just to print a message:

```
from sklearn_genetic.callbacks.base import BaseCallback

class DummyThreshold(BaseCallback):
    def __init__(self, threshold, N, metric='fitness'):
        self.threshold = threshold
        self.N = N
        self.metric = metric
```

(continues on next page)

(continued from previous page)

```
    def on_start(self, estimator=None):
        print("This training is starting!")

    def on_step(self, record, logbook, estimator=None):
        # Not enough data points
        if len(logbook) <= self.N:
            return False
        # Get the last N metrics
        stats = logbook.select(self.metric)[(-self.N - 1):]

        n_met_condition = [x for x in stats if x < self.threshold]

        if len(n_met_condition) > self.N:
            return True

        return False

    def on_end(self, logbook=None, estimator=None):
        print("I'm done with training!")
```

So that is it, now you can initialize the DummyThreshold and pass it to in the `fit` method of a *GASearchCV* instance:

```
callback = DummyThreshold(threshold=0.85, N=4, metric='fitness')
evolved_estimator.fit(X, y, callbacks=callback)
```

Here there is an output example of this callback:

```
evolved_estimator.fit(X_train, y_train, callbacks=callback)

This training is starting!
gen     nevals  fitness         fitness_std     fitness_max     fitness_min
0       16      0.339464        0.189768        0.718204        0.188695
1       14      0.501922        0.17686         0.727348        0.196176
2       16      0.630507        0.10777         0.724855        0.420615
3       10      0.718049        0.0158518       0.743142        0.669992
4       16      0.72153         0.00922375      0.735661        0.700748
INFO: Stopping the algorithm
I'm done with training!
```

Notice that there is an extra INFO message, that is general for all the callbacks that stops the training.

1.4 Using Adapters

1.4.1 Introduction

You can define adapters to have a dynamic mutation and crossover probabilities over the optimization instead of a fixed value. The idea is to make these probabilities a function of the generations; this definition can enable different training strategies, for example:

- Start with a high probability mutation to explore more diverse solutions and slowly reduce it to stay with the more promising ones.

- Start with a low crossover and end with a higher probability
- Combine different strategies for each parameter

All the methods uses three parameters:

- **initial_value:** This is the value used at generation 0
- **end_value:** It's the limit value that the parameter can take, starting from initial_value
- **adaptive_rate**: Controls how fast the value approaches the end_value; greater values increase the speed of convergence

For the following sections, it's important to understand this notation:

Name	Notation
initial_value	p_0
end_value	p_f
current generation	t
adaptive_rate	α
value at generation t	$p(t;\alpha)$

Note that p_0 doesn't need to be greater than p_f.

If $p_0 > p_f$, you are performing a decay towards p_f.

If $p_0 < p_f$, you are performing an ascend towards p_f.

All the non-constant adapters $p(t;\alpha)$, for $\alpha \in (0, 1)$, have the following properties:

$$\lim_{t->0^+} p(t;\alpha) = p_0$$

$$\lim_{t->+\infty} p(t;\alpha) = p_f$$

The following adapters are available:

- ConstantAdapter
- ExponentialAdapter
- InverseAdapter
- PotentialAdapter

1.4.2 ConstantAdapter

This adapter is meant to be used internally by the package; when the user doesn't create an adapter but instead defines the crossover or mutation probability as a real number, the package will convert it to a *ConstantAdapter*, so the library can use the internal API with the same methods in both cases. Because of this, its definition is:

$$p(t;\alpha) = p_0$$

1.4.3 ExponentialAdapter

The Exponential Adapter uses the following form to change the initial value

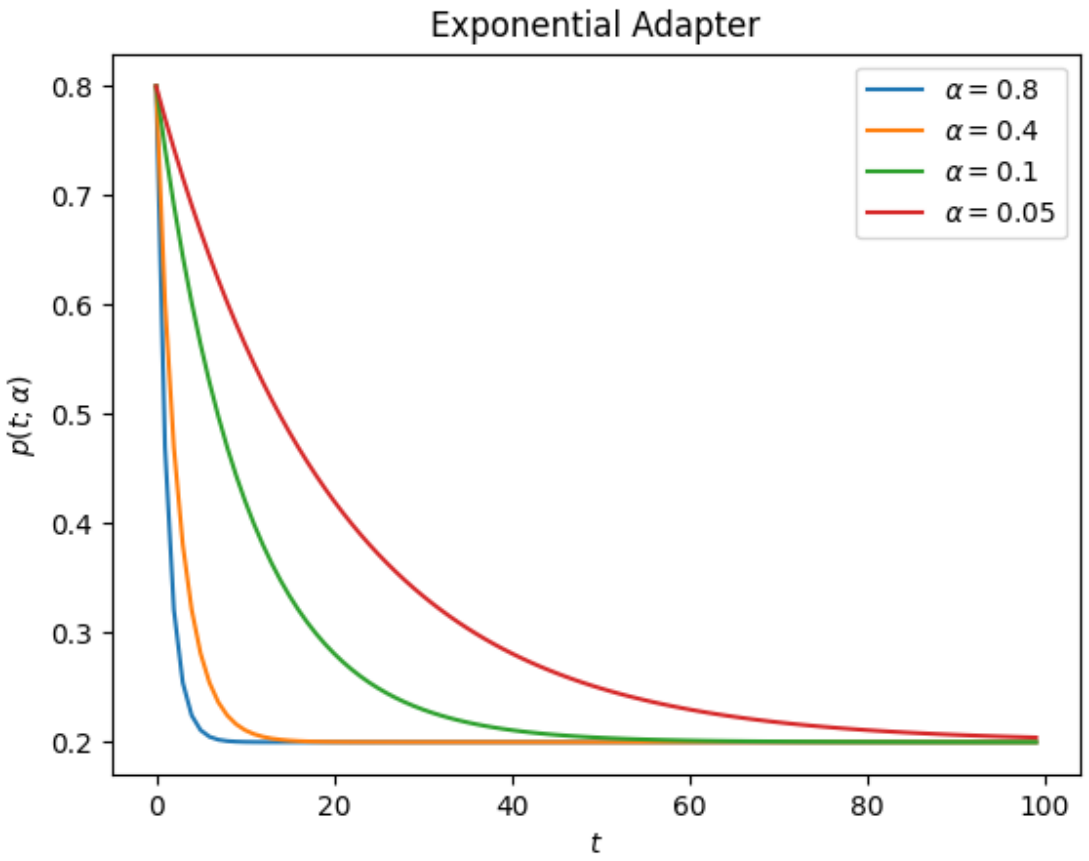
$$p(t;\alpha) = (p_0 - p_f)e^{-\alpha t} + p_f$$

Usage example:

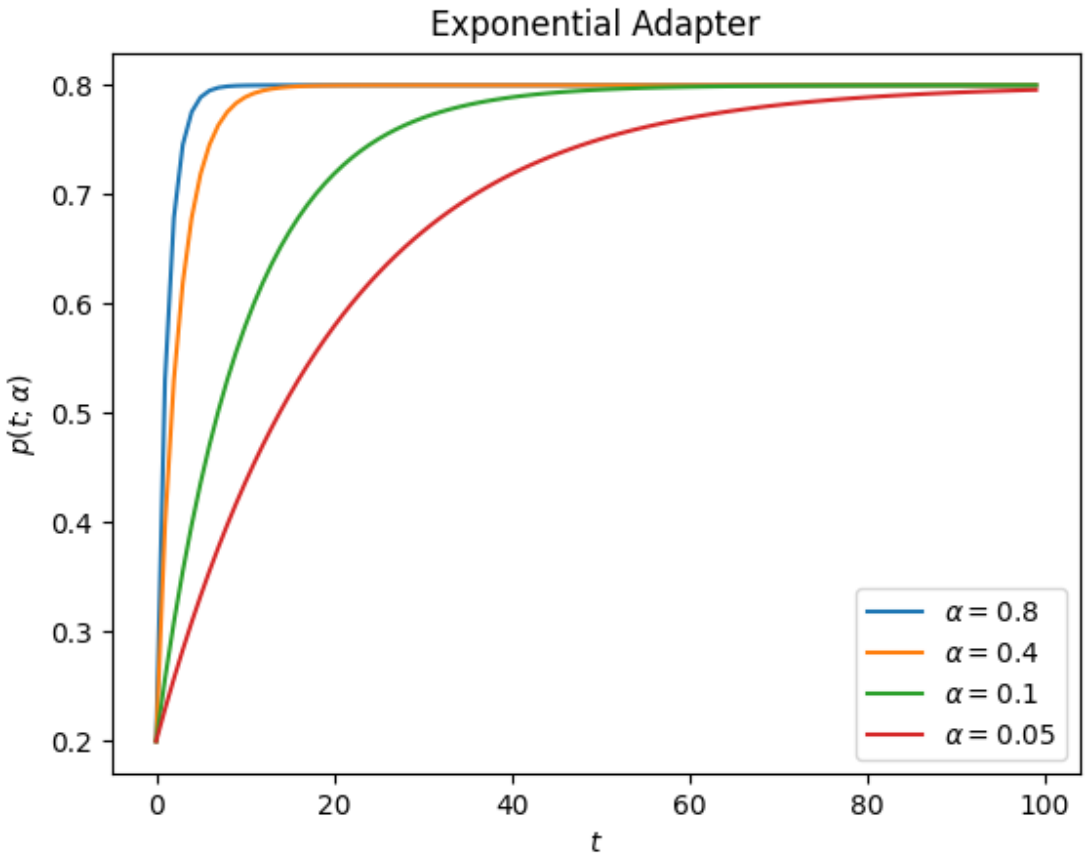
```
from sklearn_genetic.schedules import ExponentialAdapter

# Decay over initial_value
adapter = ExponentialAdapter(initial_value=0.8, end_value=0.2, adaptive_rate=0.1)

# Run a few iterations
for _ in range(3):
    adapter.step()  # 0.8, 0.74, 0.69
```

This is how the adapter looks for different values of alpha

decay:



ascend:




```
import matplotlib.pyplot as plt
from sklearn_genetic.schedules import ExponentialAdapter

values = [{"initial_value": 0.8, "end_value": 0.2},  # Decay
          {"initial_value": 0.2, "end_value": 0.8}]  # Ascend
alphas = [0.8, 0.4, 0.1, 0.05]

for value in values:
    for alpha in alphas:
        adapter = ExponentialAdapter(**value, adaptive_rate=alpha)
        adapter_result = [adapter.step() for _ in range(100)]

        plt.plot(adapter_result, label=r'$\alpha={}$'.format(alpha))

    plt.xlabel(r'$t$')
    plt.ylabel(r'$p(t; \alpha)$')
    plt.title("Exponential Adapter")
    plt.legend()
    plt.show()
```

1.4.4 InverseAdapter

The Inverse Adapter uses the following form to change the initial value

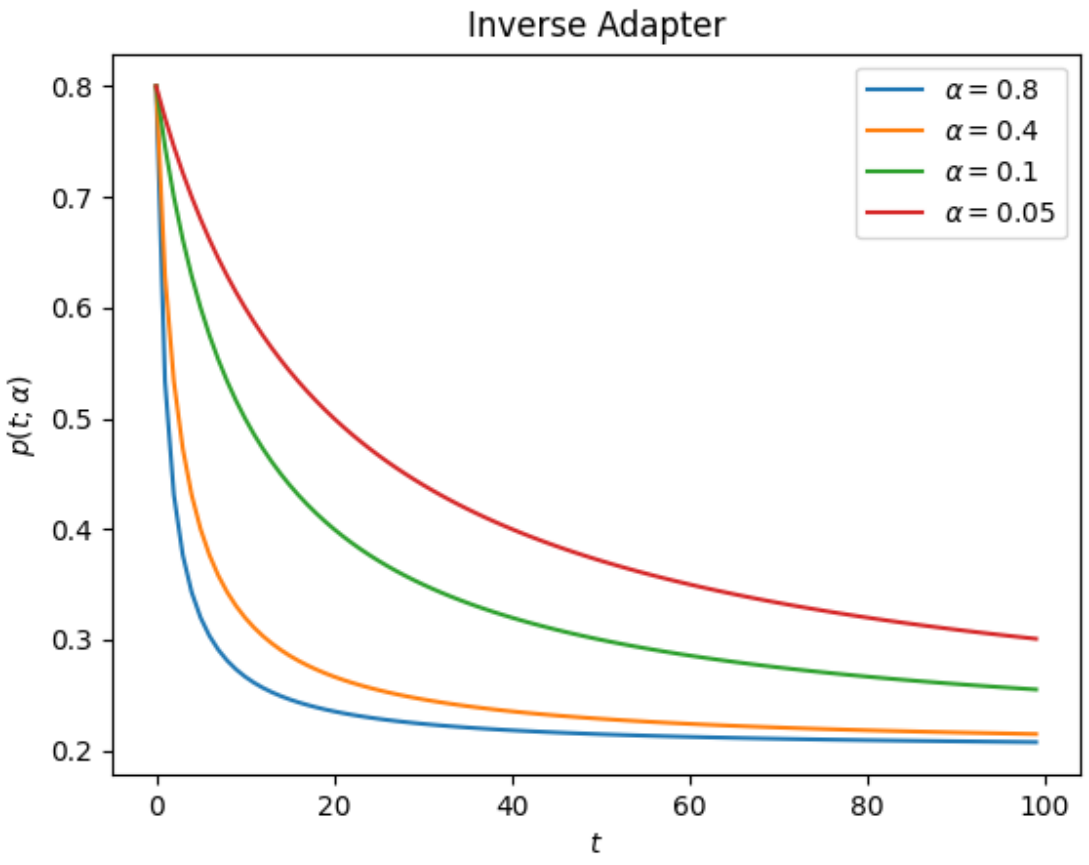
$$p(t;\alpha) = \frac{(p_0 - p_f)}{1 + \alpha t} + p_f$$

Usage example:

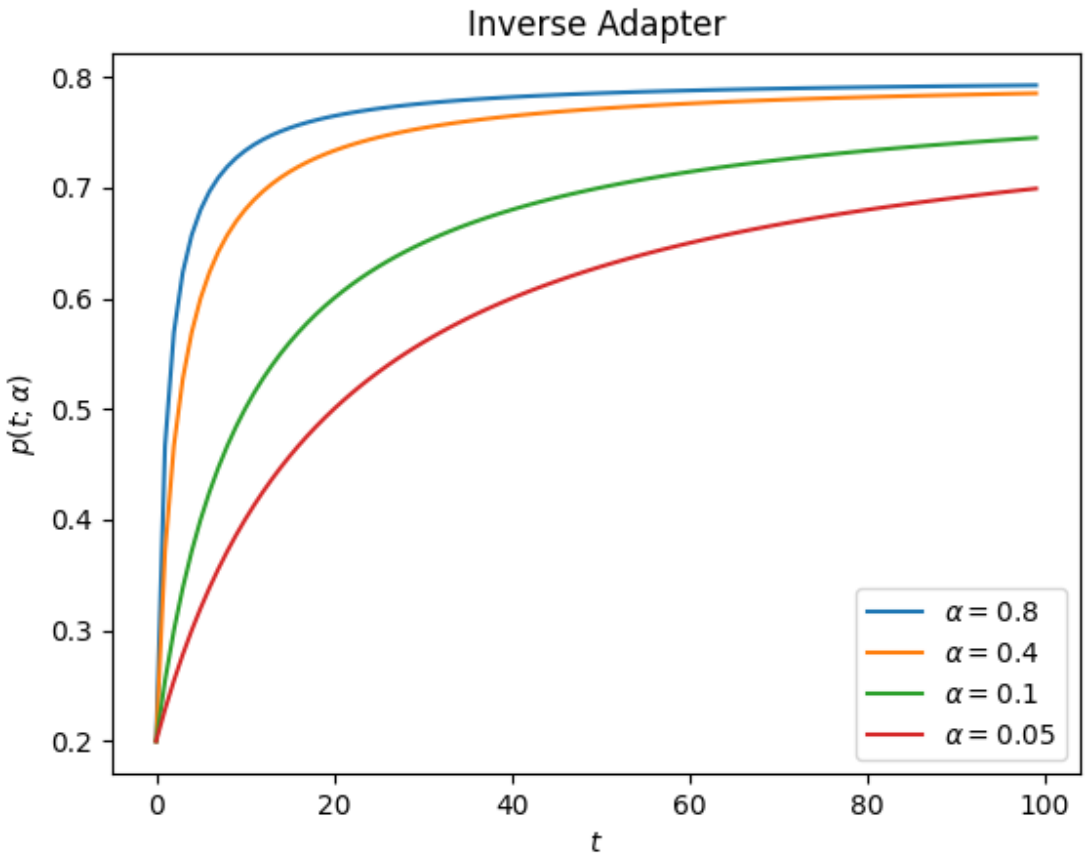
```
from sklearn_genetic.schedules import InverseAdapter

# Decay over initial_value
adapter = InverseAdapter(initial_value=0.8, end_value=0.2, adaptive_rate=0.1)

# Run a few iterations
for _ in range(3):
    adapter.step()  # 0.8, 0.75, 0.7
```

This is how the adapter looks for different values of alpha

decay:



ascend:



1.4.5 PotentialAdapter

The Inverse Adapter uses the following form to change the initial value

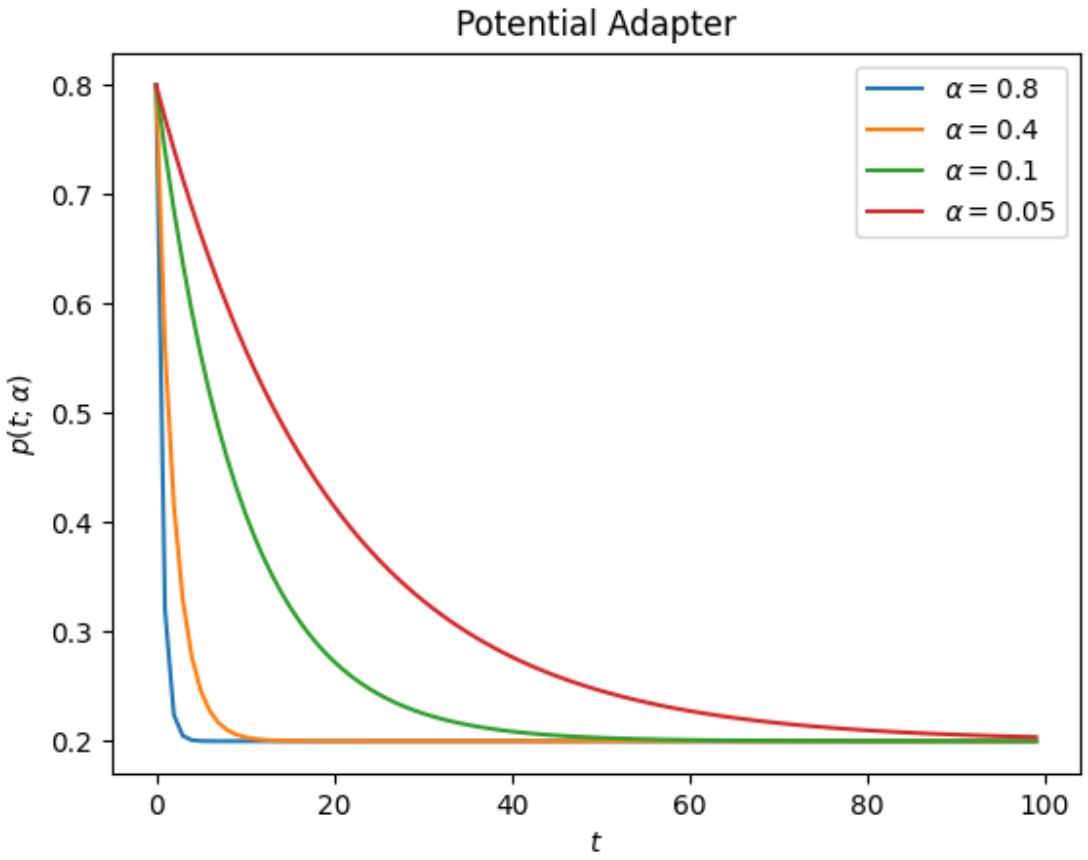
$$p(t;\alpha) = (p_0 - p_f)(1-\alpha)^t + p_f$$

Usage example:

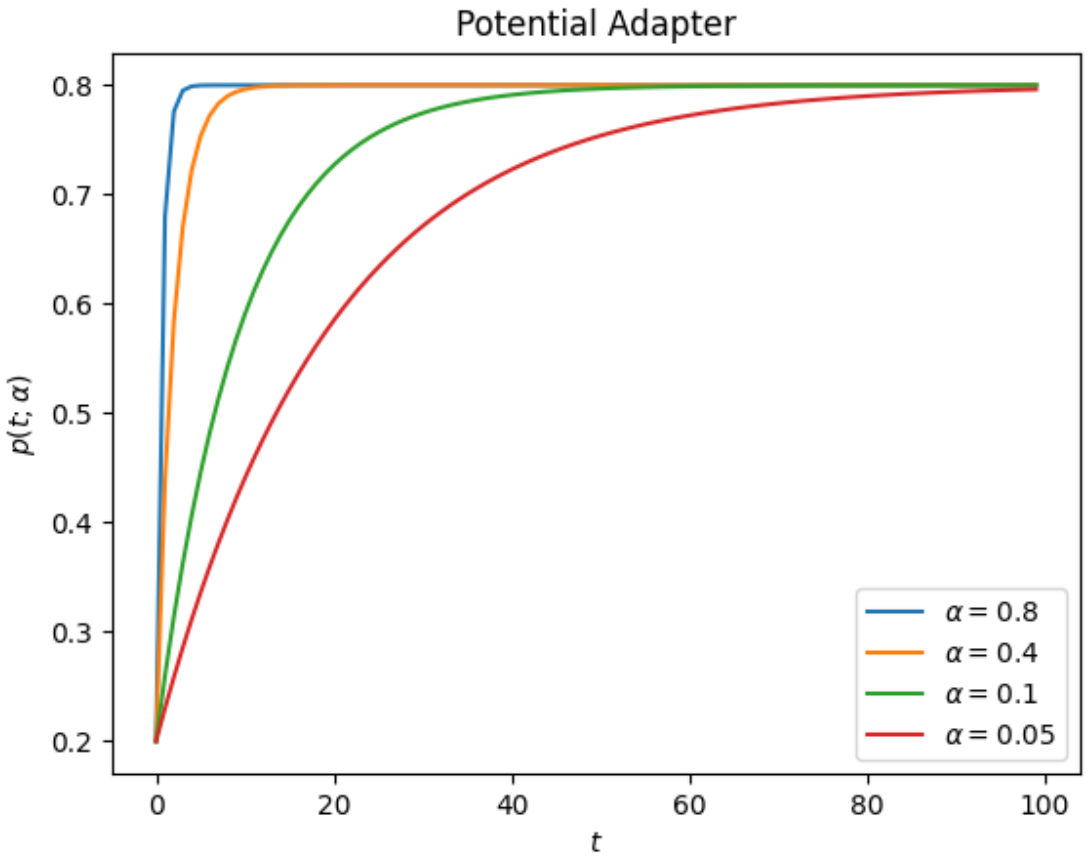
```
from sklearn_genetic.schedules import PotentialAdapter

# Decay over initial_value
adapter = PotentialAdapter(initial_value=0.8, end_value=0.2, adaptive_rate=0.1)

# Run a few iterations
for _ in range(3):
    adapter.step()  # 0.8, 0.26, 0.206
```

This is how the adapter looks for different values of alpha

decay:


Potential Adapter
$\alpha = 0.8$
$\alpha = 0.4$
$\alpha = 0.1$
$\alpha = 0.05$
$p(t; \alpha)$
t
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0
20
40
60
80
100

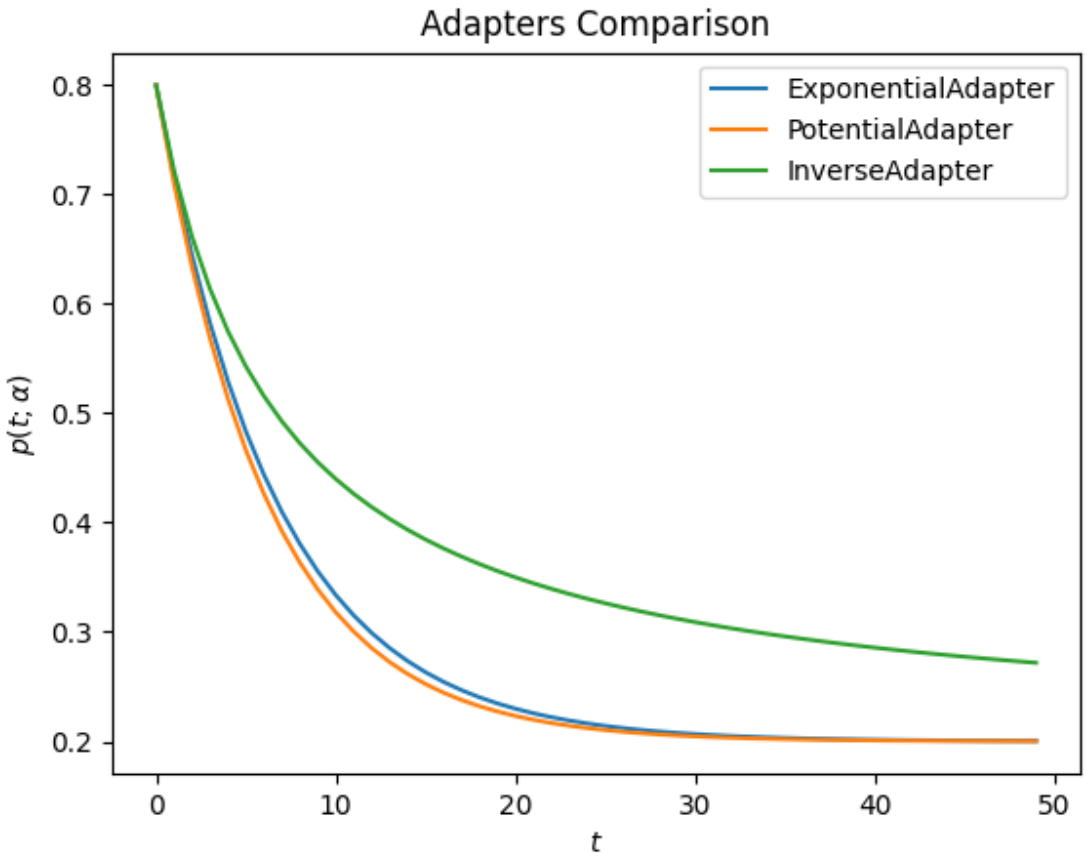
ascend:



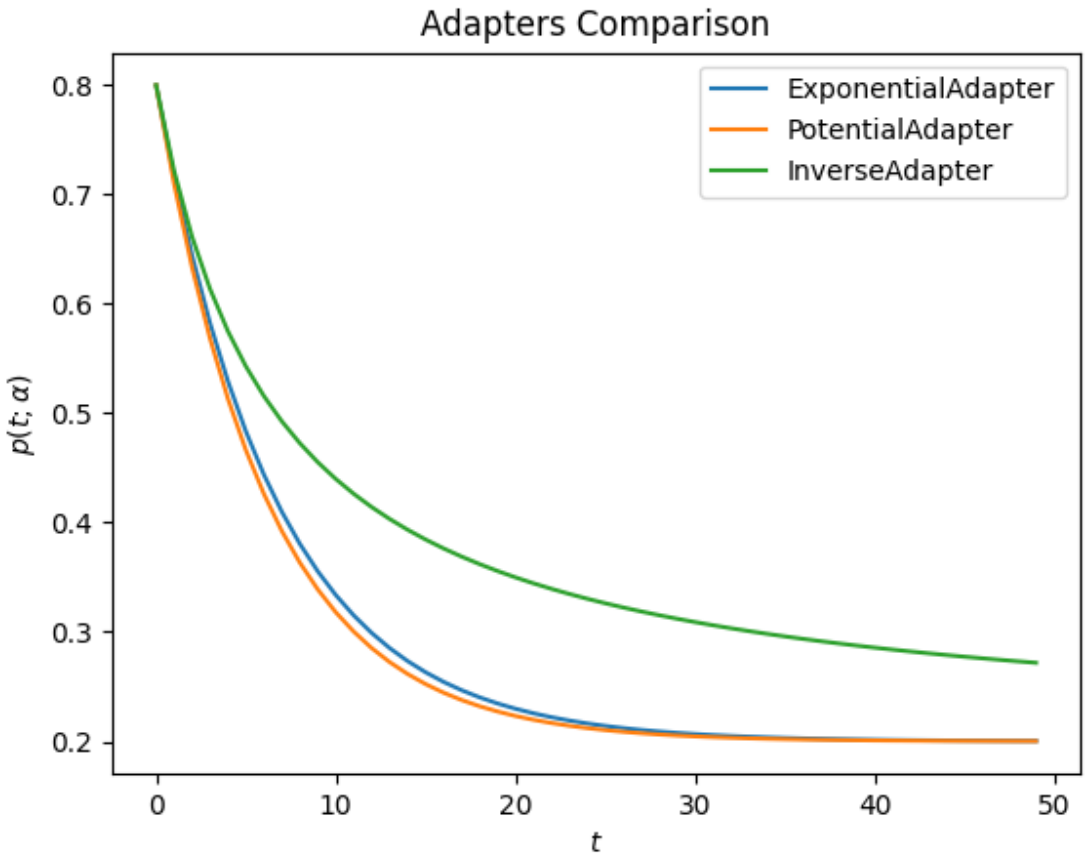
1.4.6 Compare

This is how all adapters looks like for the same value of alpha

decay:


Adapters Comparison
ExponentialAdapter
PotentialAdapter
InverseAdapter
0.8
0.7
0.6
0.5
0.4
0.3
0.2
$p(t; \alpha)$
0
10
20
30
40
50
t

ascend:




```
import matplotlib.pyplot as plt
from sklearn_genetic.schedules import ExponentialAdapter, PotentialAdapter,␣
↪InverseAdapter


params = {"initial_value": 0.2, "end_value": 0.8, "adaptive_rate": 0.15}  # Ascend
adapters = [ExponentialAdapter(**params), PotentialAdapter(**params),␣
↪InverseAdapter(**params)]

for adapter in adapters:
    adapter_result = [adapter.step() for _ in range(50)]

    plt.plot(adapter_result, label=f"{type(adapter).__name__}")

plt.xlabel(r'$t$')
plt.ylabel(r'$p(t; \alpha)$')
plt.title("Adapters Comparison")
plt.legend()
plt.show()
```

1.4.7 Full Example

In this example, we want to create a decay strategy for the mutation probability, and an ascend strategy for the crossover probability, lets call them $p_{mt}(t;\alpha)$ and $p_{cr}(t;\alpha)$ respectively; this will enable the optimizer to explore more diverse solutions in the first iterations. Take into account that on this scenario, we must be careful on choosing α, p_0, p_f, this is because the evolutionary implementation requires that:

$$p_{mt}(t;\alpha) + p_{cr}(t;\alpha) <= 1; \forall t$$

The same idea can be used for hypeparameter tuning or feature selection.

```
from sklearn_genetic import GASearchCV
from sklearn_genetic import ExponentialAdapter
from sklearn_genetic.space import Continuous, Categorical, Integer
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.datasets import load_digits
from sklearn.metrics import accuracy_score

data = load_digits()
n_samples = len(data.images)
X = data.images.reshape((n_samples, -1))
y = data['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_
↪state=42)

clf = RandomForestClassifier()

mutation_adapter = ExponentialAdapter(initial_value=0.8, end_value=0.2, adaptive_rate=0.
↪1)
crossover_adapter = ExponentialAdapter(initial_value=0.2, end_value=0.8, adaptive_rate=0.
↪1)

param_grid = {'min_weight_fraction_leaf': Continuous(0.01, 0.5, distribution='log-uniform
↪'),
              'bootstrap': Categorical([True, False]),
              'max_depth': Integer(2, 30),
              'max_leaf_nodes': Integer(2, 35),
              'n_estimators': Integer(100, 300)}

cv = StratifiedKFold(n_splits=3, shuffle=True)

evolved_estimator = GASearchCV(estimator=clf,
                               cv=cv,
                               scoring='accuracy',
                               population_size=20,
                               generations=25,
                               mutation_probability=mutation_adapter,
                               crossover_probability=crossover_adapter,
                               param_grid=param_grid,
                               n_jobs=-1)

# Train and optimize the estimator
evolved_estimator.fit(X_train, y_train)
```

(continues on next page)

(continued from previous page)

```
# Best parameters found
print(evolved_estimator.best_params_)
# Use the model fitted with the best parameters
y_predict_ga = evolved_estimator.predict(X_test)
print(accuracy_score(y_test, y_predict_ga))

# Saved metadata for further analysis
print("Stats achieved in each generation: ", evolved_estimator.history)
print("Best k solutions: ", evolved_estimator.hof)
```

1.5 Understanding the evaluation process

In this post, we are going to explain how the evaluation process works on hyperparameters tuning and how to use different validation strategies.

1.5.1 Parameters

The *GASearchCV* class, expects a parameter named *cv*. This stands for cross-validation and it accepts any of the scikit-learn strategies, such as K-fold, Repeated K-Fold, Stratified k-fold, and so on. You can find more about this in scikit-learn documentation.

A second parameter that comes along, is the *scoring*, this is the evaluation metric that the model is going to use, to decide which model is better, it could, for example be accuracy, precision, recall for a classification problem or r2, max_error, neg_root_mean_squared_error for a regression problem. To see the full list of metrics, check in here

1.5.2 Evolutionary Algorithms background

The Genetic algorithm (GA) is a metaheuristic process inspired by natural selection, it's used in optimization and search problems in general, and is usually based on a set of functions such as mutation, crossover and selection, let's call these the genetic operators. I'll use the following terms interchangeably in this section to make the connection between the GA and machine learning:

One choice of hyperparameters→An individual, Population→ Several individuals, Generation→One fixed iteration that contains a fixed population, Fitness value→Cross-validation score.

There are several variations, but in general, the steps to follow look like this:

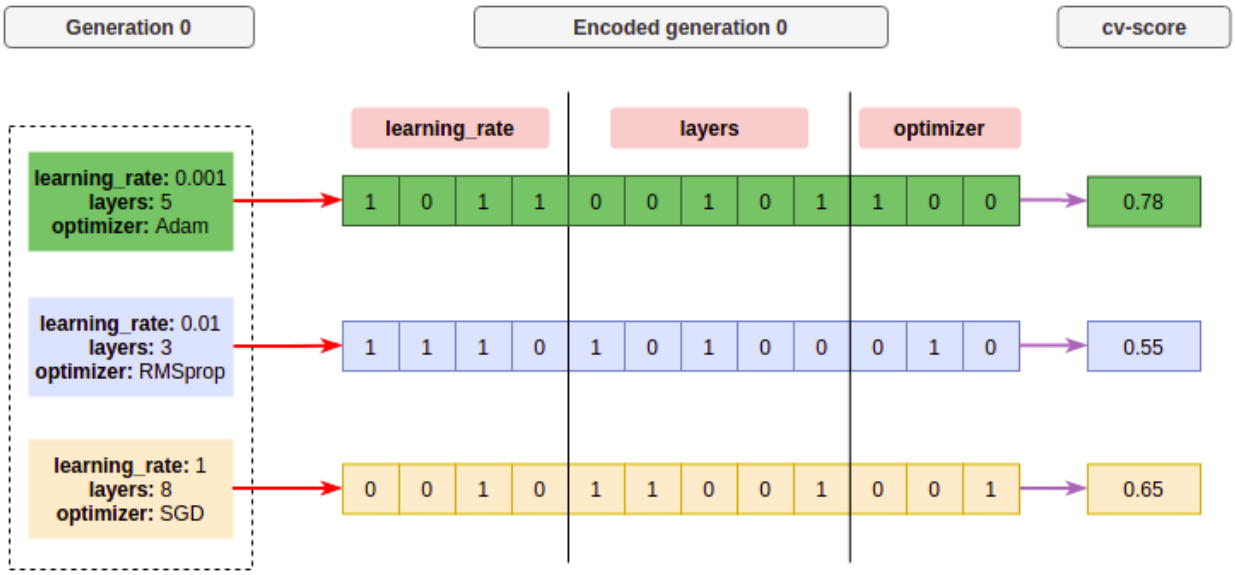
1. Generate a randomly sampled population (different sets of hyperparameters); this is generation 0.
2. Evaluate the fitness value of each individual in the population, in terms of machine learning, get the cross-validation scores.
3. Generate a new generation by using several genetic operators. Repeat steps 2 and 3 until a stopping criterion is met.

Let's go step by step.

Create generation 0 and evaluate it:

As mentioned you could generate a random set of several hyperparameters, or you could include a few manually selected ones that you already tried and think are good candidates.

Each set gets usually encoded in form of a chromosome, a binary representation of the set, for example, if we set the size of the first generation to be 3 individuals, it would look like this:



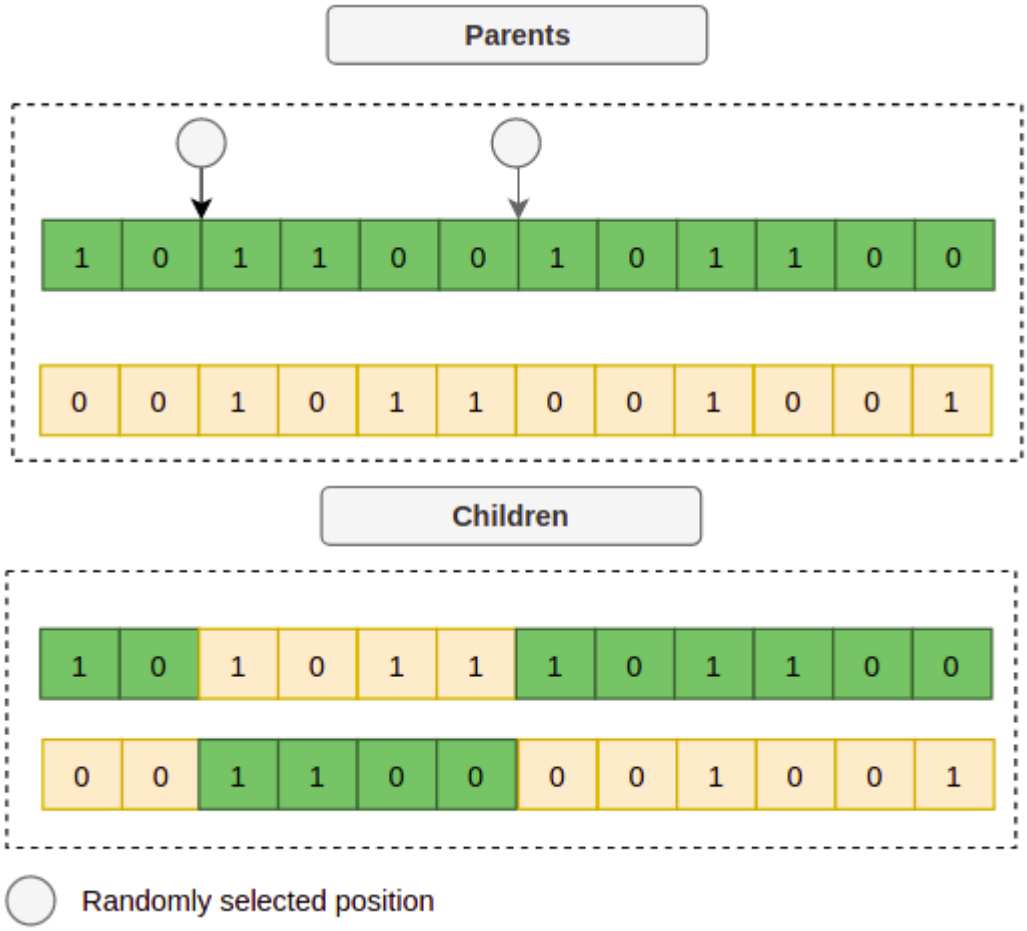
So in this generation, we get three individuals that are mapped to a chromosome (binary) representation, using an encoding function represented as the red arrow, each box in the chromosome is a gen. A fixed section of the chromosome is one of the hyperparameters. Then we get the cross-validation score (fitness) of each candidate using a scoring function, its shown as the purple arrow.

Create a new generation:

Now we can create a new set of candidates, as mentioned, there are several genetic operators, I'm going to show the most common ones:

Crossover:

This operator consists of taking two parent chromosomes and mates them to create new children, the way we select the parents could be by a probability distribution function, which gives more probability to the individuals with higher fitness of the generation, let's say the individual number 1 and 3 got selected, then we can take two random points of each parent and make the crossover, like this:



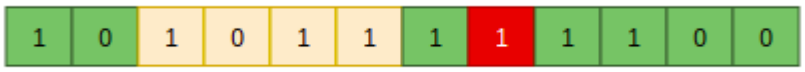
Now the children represent a new set of hyperparameters, if we decode each child we could get for example:

```
Child 1: {"learning_rate": 0.015, "layers": 4, "optimizer": "Adam"}
Child 2: {"learning_rate": 0.4, "layers": 6, "optimizer": "SGD"}
```

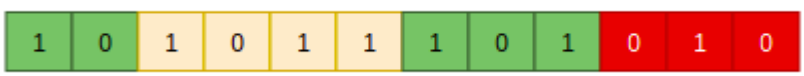
But making crossovers, over the same sets of hyperparameters might end up giving similar results after some iterations, so we are stuck with the same kind of solutions, that is why we introduce other operations like the mutation.

Mutation:

This operator allows with a low enough probability (< ~0.1), to change one of the gens or a whole hyperparameter randomly, to create more diverse sets. Let's take, for example, child 1 from the previous image, let's pick up a random gen and change its value:

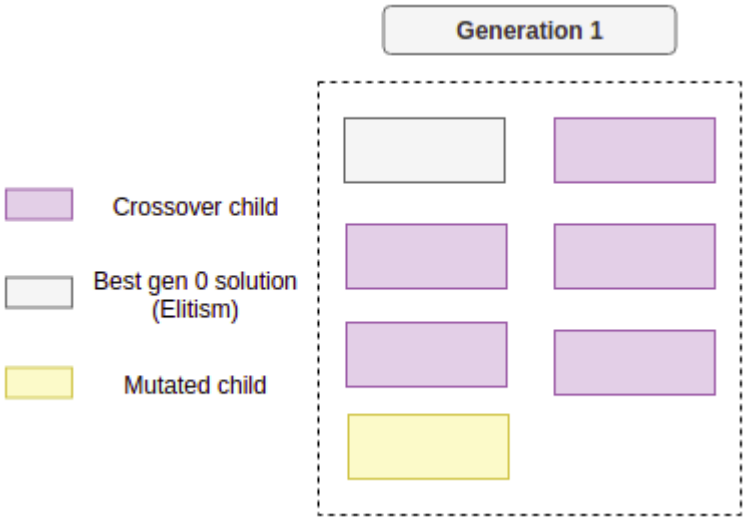


Or it could even change a whole parameter, for example, the optimizer:



Elitism:

This selection strategy refers to the process of selecting the best individuals of each generation, to make sure its information is propagated across the generations. This is very straightforward, just select the best k individuals based on their fitness value and copy it to the next generation. So after performing those operations, a new generation may look like this:



From now on, just repeat the process for several generations until a stopping criteria is met, those could be for example:

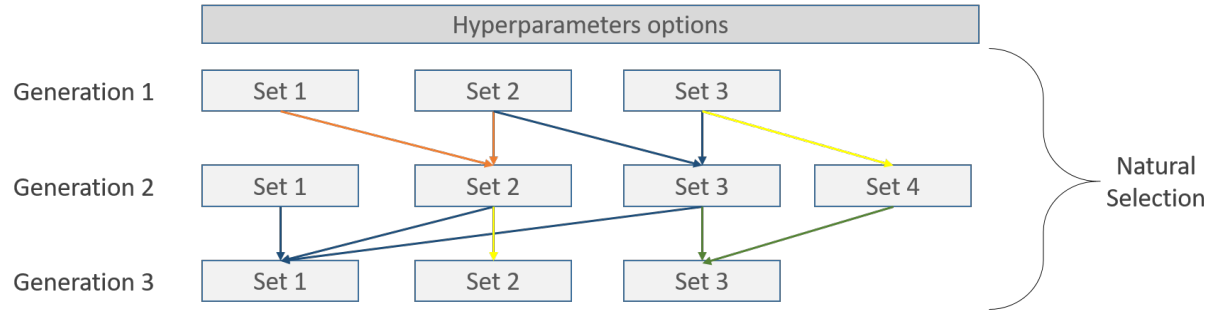
- A maximum number of generations was reached.
- The process has run longer than the budgeted time.
- There are no performance improvements (below a threshold) from the last n generations.

1.5.3 Steps

Now, moving to this package implementation. The way *GASearchCV* evaluates the candidates is as follows:

- It starts by selecting random sets of hyperparameters according to the *param_grid* definition, the total number of sets is determined by the *population_size* parameter.
- It fits a model per each sets of hyperparameters and calculates the cross validation score according to the *cv* and *scoring* setup.
- After evaluating each candidate, the fitness, fitness_std, fitness_max and fitness_min are computed and are logged into the console if `verbose=True`. *Fitness* is the way to refer to the selected metric, but this is calculated as the average of all the candidates of the current generation, this means that if there are 10 different sets of hyperparameters, the *fitness* value, is the average score of those 10 evaluated candidates, the same goes for the other metrics.
- Now it creates new sets (generations) of hyperparameters, those are created by combining the last generation with different strategies, those strategies depends on the selected `algorithms`.
- It repeats steps 2, 3 and 4 until the number of generations is met, or until callbacks stop the process.
- At the end, the algorithm selects the best hyperparameters, as the set that got the best individual cross-validation scoring.

Those steps could be represented like this, each line represents one of several possible natural processes like mating, crossover, selection and mutation:



Inside each set, the cross validation takes place, for example, using the 5-Folds strategy

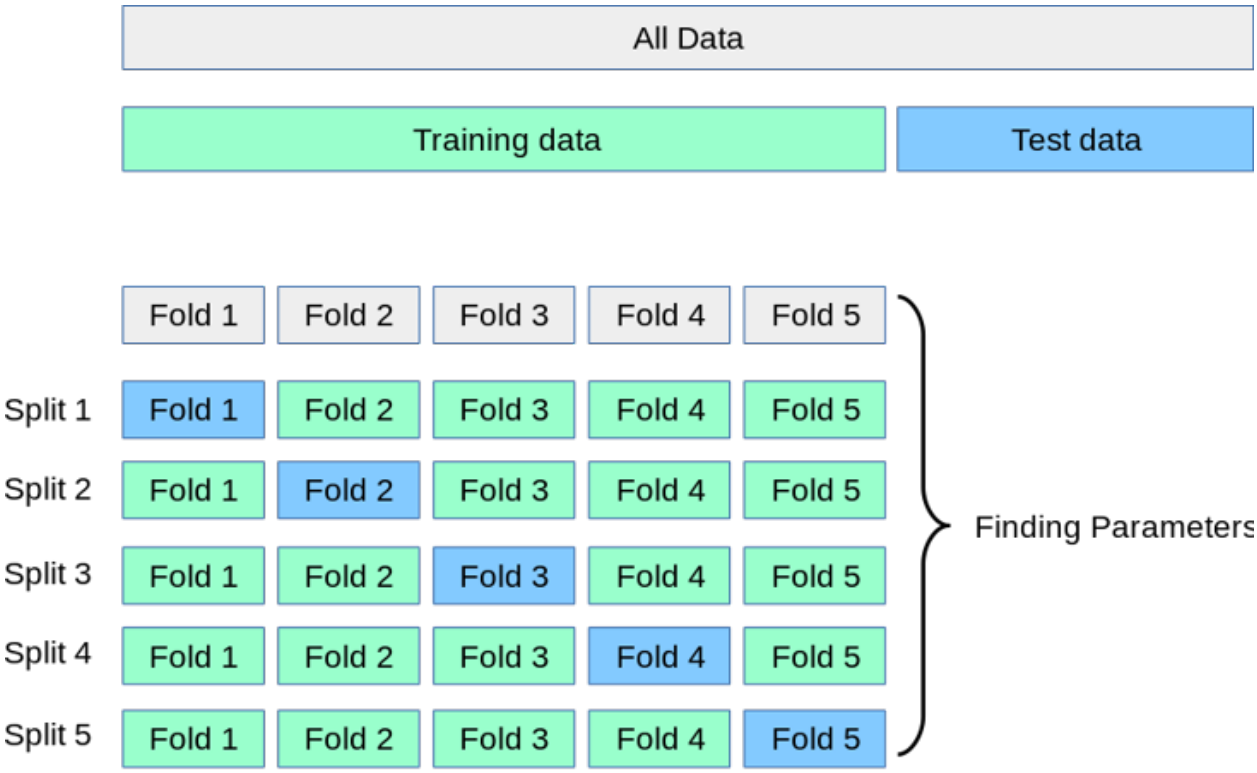


Image is taken from scikit-learn

1.5.4 Example

This example is going to use a regression problem from the Boston house prices dataset. We are going to use a K-Fold with 5 splits taking as evaluation the r-squared metric.

In the end, we are going to print the top 4 solutions and the r-squared on the test set for the best set of hyperparameters.

```
from sklearn_genetic import GASearchCV
from sklearn_genetic.space import Integer, Categorical, Continuous
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split, KFold
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import r2_score
from sklearn.pipeline import Pipeline
```

(continues on next page)

(continued from previous page)

```
from sklearn.preprocessing import StandardScaler

data = load_boston()

y = data["target"]
X = data["data"]

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.33, random_state=42)

cv = KFold(n_splits=5, shuffle=True)

clf = DecisionTreeRegressor()

pipe = Pipeline([('scaler', StandardScaler()), ('clf', clf)])

param_grid = {
    "clf__ccp_alpha": Continuous(0, 1),
    "clf__criterion": Categorical(["mse", "mae"]),
    "clf__max_depth": Integer(2, 20),
    "clf__min_samples_split": Integer(2, 30),
}

evolved_estimator = GASearchCV(
    estimator=pipe,
    cv=3,
    scoring="r2",
    population_size=15,
    generations=20,
    tournament_size=3,
    elitism=True,
    keep_top_k=4,
    crossover_probability=0.9,
    mutation_probability=0.05,
    param_grid=param_grid,
    criteria="max",
    algorithm="eaMuCommaLambda",
    n_jobs=-1,
)

evolved_estimator.fit(X_train, y_train)
y_predict_ga = evolved_estimator.predict(X_test)
r_squared = r2_score(y_test, y_predict_ga)

print(evolved_estimator.best_params_)
print("r-squared: ", "{:.2f}".format(r_squared))
```

1.6 Integrating with MLflow

In this post, we are going to explain how setup the build-in integration of sklearn-genetic-opt with MLflow. To use this feature, we must set the parameters that will include the tracking server, experiment name, run name, tags and others, the full implementation is here: *MLflowConfig*

1.6.1 Configuration

The configuration is pretty straightforward, we just need to import the main class and define some parameters, here there is its meaning:

- **tracking_uri:** Address of local or remote-tracking server.
- **experiment:** Case sensitive name of an experiment to be activated.
- **run_name:** Name of new run (stored as a mlflow.runName tag).
- **save_models:** If `True`, it will log the estimator into mlflow artifacts.
- **registry_uri:** Address of local or remote model registry server.
- **tags:** Dictionary of tags to apply.

1.6.2 Example

In this example, we are going to log the information into a mlflow server that is running in our localhost, port 5000, we want to save each of the trained models.

```
from sklearn_genetic.mlflow_log import MLflowConfig

mlflow_config = MLflowConfig(
    tracking_uri="http://localhost:5000",
    experiment="Digits-sklearn-genetic-opt",
    run_name="Decision Tree",
    save_models=True,
    tags={"team": "sklearn-genetic-opt", "version": "0.5.0"})
```

Now, this config is passed to the *GASearchCV* class in the parameter named *log_config*, for example:

```
from sklearn_genetic import GASearchCV
from sklearn_genetic.space import Categorical, Integer, Continuous
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_digits
from sklearn.metrics import accuracy_score
from sklearn_genetic.mlflow import MLflowConfig


data = load_digits()
label_names = data["target_names"]
y = data["target"]
X = data["data"]

X_train, X_test, y_train, y_test = train_test_split(
```

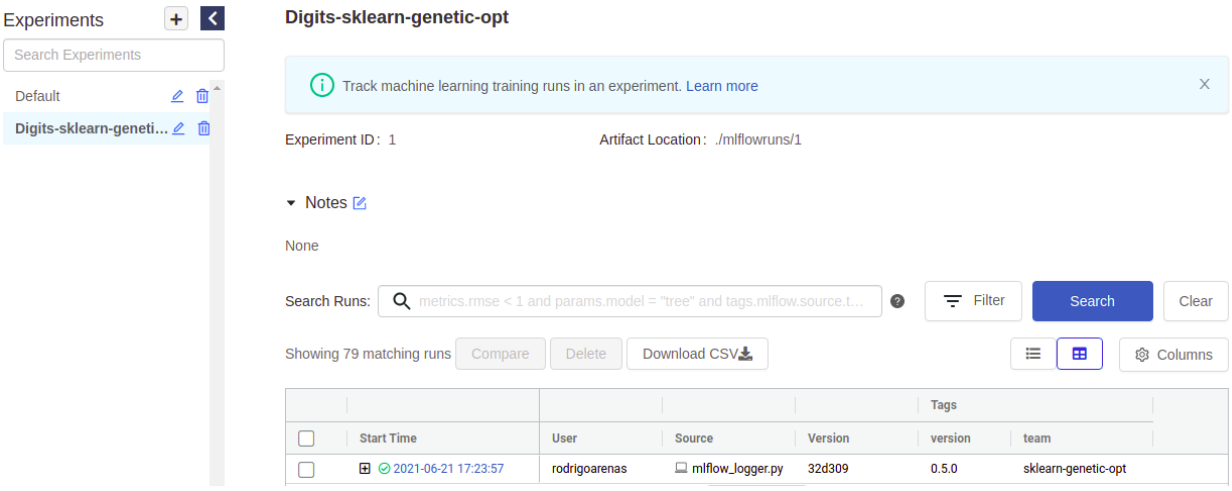
(continues on next page)

(continued from previous page)

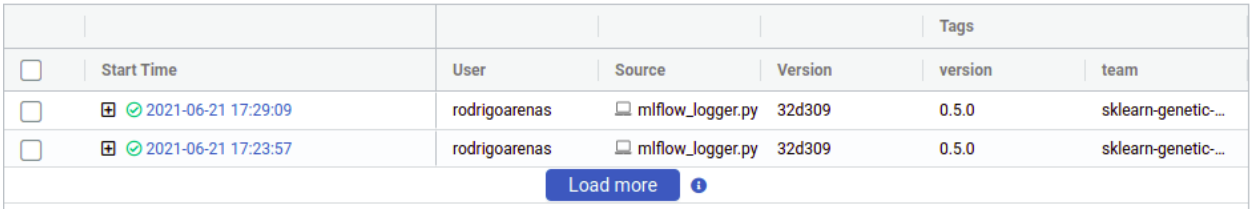
```
    X, y, test_size=0.33, random_state=42)

clf = DecisionTreeClassifier()

params_grid = {
    "min_weight_fraction_leaf": Continuous(0, 0.5),
    "criterion": Categorical(["gini", "entropy"]),
    "max_depth": Integer(2, 20),
    "max_leaf_nodes": Integer(2, 30)}

cv = StratifiedKFold(n_splits=3, shuffle=True)

evolved_estimator = GASearchCV(
    clf,
    cv=cv,
    scoring="accuracy",
    population_size=3,
    generations=5,
    tournament_size=3,
    elitism=True,
    crossover_probability=0.9,
    mutation_probability=0.05,
    param_grid=params_grid,
    algorithm="eaMuPlusLambda",
    n_jobs=-1,
    verbose=True,
    log_config=mlflow_config)

evolved_estimator.fit(X_train, y_train)
y_predict_ga = evolved_estimator.predict(X_test)
accuracy = accuracy_score(y_test, y_predict_ga)

print(evolved_estimator.best_params_)
```

Notice that we choose small generations and population_size, just to be able to see the results without much verbosity.

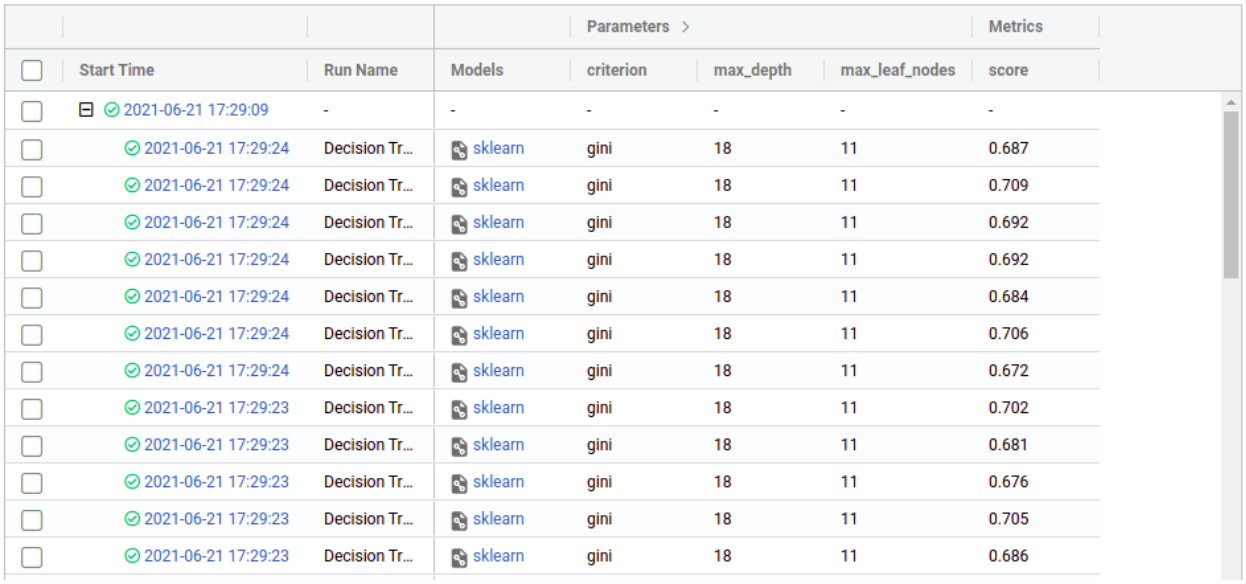
If you go to your mlflow UI and click the experiment named “Digits-sklearn-genetic-opt” we should see something like this (I’ve hidden some columns to give a better look):



There we can see the user that ran the experiment, the name of the file which contained the source code, our tags and other metadata. Notice that there is a "plus" symbol that will show us each of our iterations, this is because sklearn-genetic-opt will log each *GASearchCV.fit()* call in a nested way, think it like a parent run, and each child is one of the hyperparameters that were tested, for example, if we run the same code again, now we see two parents run:



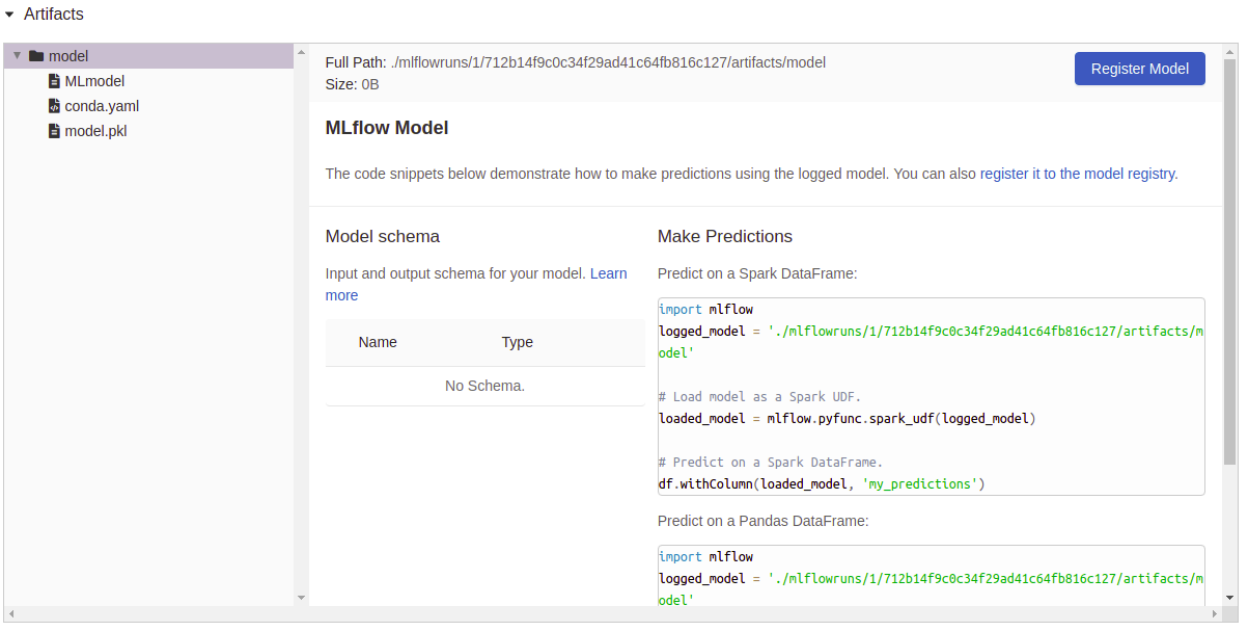
Now click on any of the "plus" symbols to see all the children, now they look like this (again edited the columns to display):



From there we can see the hyperparameters and the score (cross-validation) that we got in each run, from there we can use the regular mlflow functionalities like comparing runs, download the CSV, register a model, etc. You can see more on https://mlflow.org/docs/latest/index.html

Now, as we set `save_model=True`, you can see that the column "Model" has a file attached as an artifact, if we click

on one of those, we see a resume of that particular execution and some utils to use right away the model:



1.7 Reproducibility

One of the desirable capabilities of a package that makes several "random" choices is to be able to reproduce the results.

The usual strategy is to fix the random seed that starts generating the pseudo-random numbers. Unfortunately, the DEAP package, which is the main dependency for all the evolutionary algorithms, doesn't have an explicit parameter to fix this seed.

However, there is a workaround that seems to work to reproduce these results; this is:

- Set the random seed of *numpy* and *random* package, which are the underlying random numbers generators
- Use the random_state parameter In each of the scikit-learn and sklearn-genetic-opt objects that support it

In the following example, the random_state is set for the *train_test_split*, *cross-validation* generator, each of the hyper-parameters in the *param_grid*, the *RandomForestClassifier*, and at the file level.

1.7.1 Example:

```
import numpy as np
import random
from sklearn_genetic import GASearchCV
from sklearn_genetic.space import Continuous, Categorical, Integer
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.datasets import load_digits
from sklearn.metrics import accuracy_score


# Random Seed at file level
```

(continues on next page)

(continued from previous page)

```
random_seed = 54

np.random.seed(random_seed)
random.seed(random_seed)


data = load_digits()
n_samples = len(data.images)
X = data.images.reshape((n_samples, -1))
y = data['target']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_
↪state=random_seed)

clf = RandomForestClassifier(random_state=random_seed)

param_grid = {'min_weight_fraction_leaf': Continuous(0.01, 0.5, distribution='log-uniform
↪',
                                                     random_state=random_seed),
              'bootstrap': Categorical([True, False], random_state=random_seed),
              'max_depth': Integer(2, 30, random_state=random_seed),
              'max_leaf_nodes': Integer(2, 35, random_state=random_seed),
              'n_estimators': Integer(100, 300, random_state=random_seed)}

cv = StratifiedKFold(n_splits=3, shuffle=True, random_state=random_seed)

evolved_estimator = GASearchCV(estimator=clf,
                               cv=cv,
                               scoring='accuracy',
                               population_size=8,
                               generations=5,
                               param_grid=param_grid,
                               n_jobs=-1,
                               verbose=True,
                               keep_top_k=4)

# Train and optimize the estimator
evolved_estimator.fit(X_train, y_train)
# Best parameters found
print(evolved_estimator.best_params_)
# Use the model fitted with the best parameters
y_predict_ga = evolved_estimator.predict(X_test)
print(accuracy_score(y_test, y_predict_ga))

# Saved metadata for further analysis
print("Stats achieved in each generation: ", evolved_estimator.history)
print("Best k solutions: ", evolved_estimator.hof)
```

1.8 Scikit-learn Comparison

[1]:
```
from sklearn_genetic import GASearchCV
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import train_test_split, GridSearchCV, RandomizedSearchCV
from sklearn_genetic.space import Categorical, Continuous
import scipy.stats as stats
from sklearn.utils.fixes import loguniform
from sklearn.datasets import load_digits
from sklearn.metrics import accuracy_score
import numpy as np
import warnings
warnings.filterwarnings("ignore")
```

[2]:
```
data = load_digits()
```

[3]:
```
label_names = data['target_names']
y = data['target']
X = data['data']
```

[4]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_
↪state=42)
```

[5]:
```
clf = SGDClassifier(loss='hinge',fit_intercept=True)
```

1.8.1 1. Random Search

[6]:
```
param_dist = {'average': [True, False],
              'l1_ratio': stats.uniform(0, 1),
              'alpha': loguniform(1e-4, 1e0)}
```

[7]:
```
n_iter_search = 30
random_search = RandomizedSearchCV(clf, param_distributions=param_dist,
                                   n_iter=n_iter_search,n_jobs=-1)
```

[8]:
```
random_search.fit(X_train,y_train)
```

[8]:
```
RandomizedSearchCV(estimator=SGDClassifier(), n_iter=30, n_jobs=-1,
                   param_distributions={'alpha': <scipy.stats._distn_infrastructure.rv_
↪frozen object at 0x000001A62568BD60>,
                                        'average': [True, False],
                                        'l1_ratio': <scipy.stats._distn_infrastructure.
↪rv_frozen object at 0x000001A61065B400>})
```

[9]:
```
accuracy_score(y_test, random_search.predict(X_test))
```

[9]:
```
0.9629629629629629
```

[10]:
```
random_search.best_params_
```

[10]:
```
{'alpha': 0.020380435883006108,
 'average': True,
 'l1_ratio': 0.01937382409973476}
```

1.8.2 2. Grid Search

[11]:
```
param_grid = {'average': [True, False],
              'l1_ratio': np.linspace(0, 1, num=10),
              'alpha': np.power(10, np.arange(-4, 1, dtype=float))}
```

[12]:
```
grid_search = GridSearchCV(clf, param_grid=param_grid,n_jobs=-1)
```

[13]:
```
grid_search.fit(X_train,y_train)
```

[13]:
```
GridSearchCV(estimator=SGDClassifier(), n_jobs=-1,
             param_grid={'alpha': array([1.e-04, 1.e-03, 1.e-02, 1.e-01, 1.e+00]),
                         'average': [True, False],
                         'l1_ratio': array([0.        , 0.11111111, 0.22222222, 0.
↪33333333, 0.44444444,
       0.55555556, 0.66666667, 0.77777778, 0.88888889, 1.        ])})
```

[14]:
```
accuracy_score(y_test, grid_search.predict(X_test))
```

[14]:
```
0.9528619528619529
```

[15]:
```
grid_search.best_params_
```

[15]:
```
{'alpha': 0.001, 'average': True, 'l1_ratio': 0.4444444444444444}
```

1.8.3 3. Genetic Algorithm

[16]:
```
param_grid = {'l1_ratio': Continuous(0,1),
              'alpha': Continuous(1e-4,1),
              'average': Categorical([True, False])}

evolved_estimator = GASearchCV(clf,
                    cv=3,
                    scoring='accuracy',
                    param_grid=param_grid,
                    population_size=10,
                    generations=8,
                    tournament_size=3,
                    elitism=True,
                    verbose=True)
```

[17]:
```
evolved_estimator.fit(X_train,y_train)
```

```
gen     nevals  fitness         fitness_std    fitness_max    fitness_min
0       10      0.939817        0.00313682      0.945137       0.934331
1       10      0.940482        0.00433848      0.9468         0.932668
```

(continues on next page)

(continued from previous page)

```
2       10      0.940482        0.00226736      0.943475        0.935162
3       10      0.942228        0.00244479      0.945137        0.938487
4       10      0.939734        0.00420996      0.945137        0.934331
5       10      0.937323        0.00362717      0.944306        0.931837
6       10      0.943475        0.00313241      0.949293        0.939318
7       10      0.940399        0.0042394       0.950125        0.934331
8       10      0.943724        0.00257689      0.948462        0.938487
```

[17]:
```
<sklearn_genetic_opt.GASearchCV at 0x1a625628ee0>
```

[18]:
```
y_predicy_ga = evolved_estimator.predict(X_test)
```

[19]:
```
accuracy_score(y_test,y_predicy_ga)
```

[19]:
```
0.968013468013468
```

[20]:
```
evolved_estimator.best_params
```

[20]:
```
{'l1_ratio': 0.9918490625641972, 'alpha': 0.5633014570910942, 'average': False}
```

1.9 Boston House Pricing Prediction

[1]:
```
import matplotlib.pyplot as plt
from sklearn_genetic import GASearchCV
from sklearn_genetic.space import Integer, Categorical, Continuous
from sklearn_genetic.plots import plot_fitness_evolution, plot_search_space
from sklearn_genetic.callbacks import LogbookSaver, ProgressBar
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split, KFold
from sklearn.tree import DecisionTreeRegressor
from sklearn.metrics import r2_score
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
```

1.9.1 Import the data and split it in train and test sets

[2]:
```
data = load_boston()

y = data["target"]
X = data["data"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_
↪state=42)
```

1.9.2 Define the classifier to tune

[3]:
```
clf = DecisionTreeRegressor()
pipe = Pipeline([("scaler", StandardScaler()), ("clf", clf)])
```

1.9.3 Create the CV strategy and define the param grid

[4]:
```
cv = KFold(n_splits=5, shuffle=True)

param_grid = {
    "clf__ccp_alpha": Continuous(0, 1),
    "clf__criterion": Categorical(["mse", "mae"]),
    "clf__max_depth": Integer(2, 20),
    "clf__min_samples_split": Integer(2, 30)}
```

1.9.4 Define the GASearchCV options

[5]:
```
evolved_estimator = GASearchCV(
    estimator=pipe,
    cv=3,
    scoring="r2",
    population_size=15,
    generations=20,
    tournament_size=3,
    elitism=True,
    keep_top_k=4,
    crossover_probability=0.9,
    mutation_probability=0.05,
    param_grid=param_grid,
    criteria="max",
    algorithm="eaMuCommaLambda",
    n_jobs=-1)
```

1.9.5 Optionally, create some Callbacks

[6]:
```
callbacks = [LogbookSaver(checkpoint_path="./logbook.pkl"), ProgressBar()]
```

1.9.6 Fit the model and see some results

[7]:
```
evolved_estimator.fit(X_train, y_train, callbacks=callbacks)
y_predict_ga = evolved_estimator.predict(X_test)
r_squared = r2_score(y_test, y_predict_ga)
```

```
  0%|          | 0/21 [00:00<?, ?it/s]
```

```
gen     nevals  fitness         fitness_std    fitness_max    fitness_min
0       15      0.616334        0.0737326       0.699908        0.520033
```

(continues on next page)

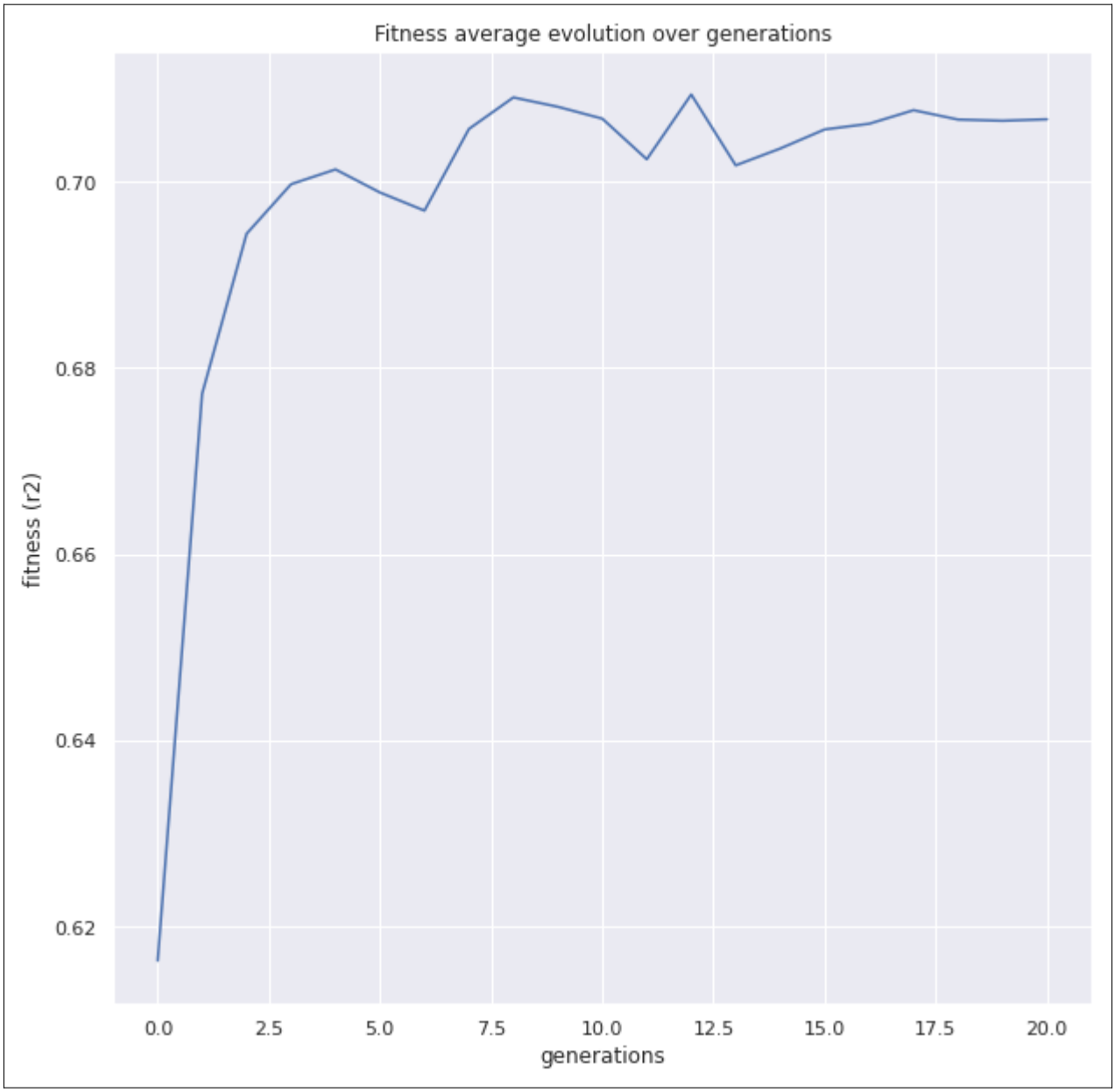
(continued from previous page)

```
1       29      0.677235        0.0439198       0.708851        0.520033
2       30      0.694427        0.0193834       0.738255        0.6749
3       29      0.699751        0.0154815       0.727109        0.676053
4       28      0.701338        0.00901281      0.713701        0.680425
5       28      0.698859        0.00957983      0.714697        0.683542
6       29      0.696912        0.0104028       0.709564        0.680759
7       26      0.705685        0.00819244      0.714573        0.683011
8       29      0.70907         0.00473398      0.714573        0.699808
9       28      0.708067        0.00616905      0.714526        0.695999
10      27      0.70679         0.00423967      0.711368        0.694636
11      30      0.702428        0.0053359       0.710461        0.695597
12      27      0.709388        0.00359735      0.713665        0.703131
13      29      0.701775        0.00600001      0.707575        0.691636
14      30      0.703581        0.00699099      0.712533        0.692699
15      29      0.705634        0.00417138      0.709928        0.692095
16      29      0.706242        0.00383751      0.709808        0.694379
17      29      0.7077          0.00347843      0.711972        0.699101
18      28      0.706694        0.00608173      0.712897        0.690798
19      28      0.706568        0.00477195      0.712003        0.69356
20      30      0.706721        0.00587165      0.71358         0.694205
```

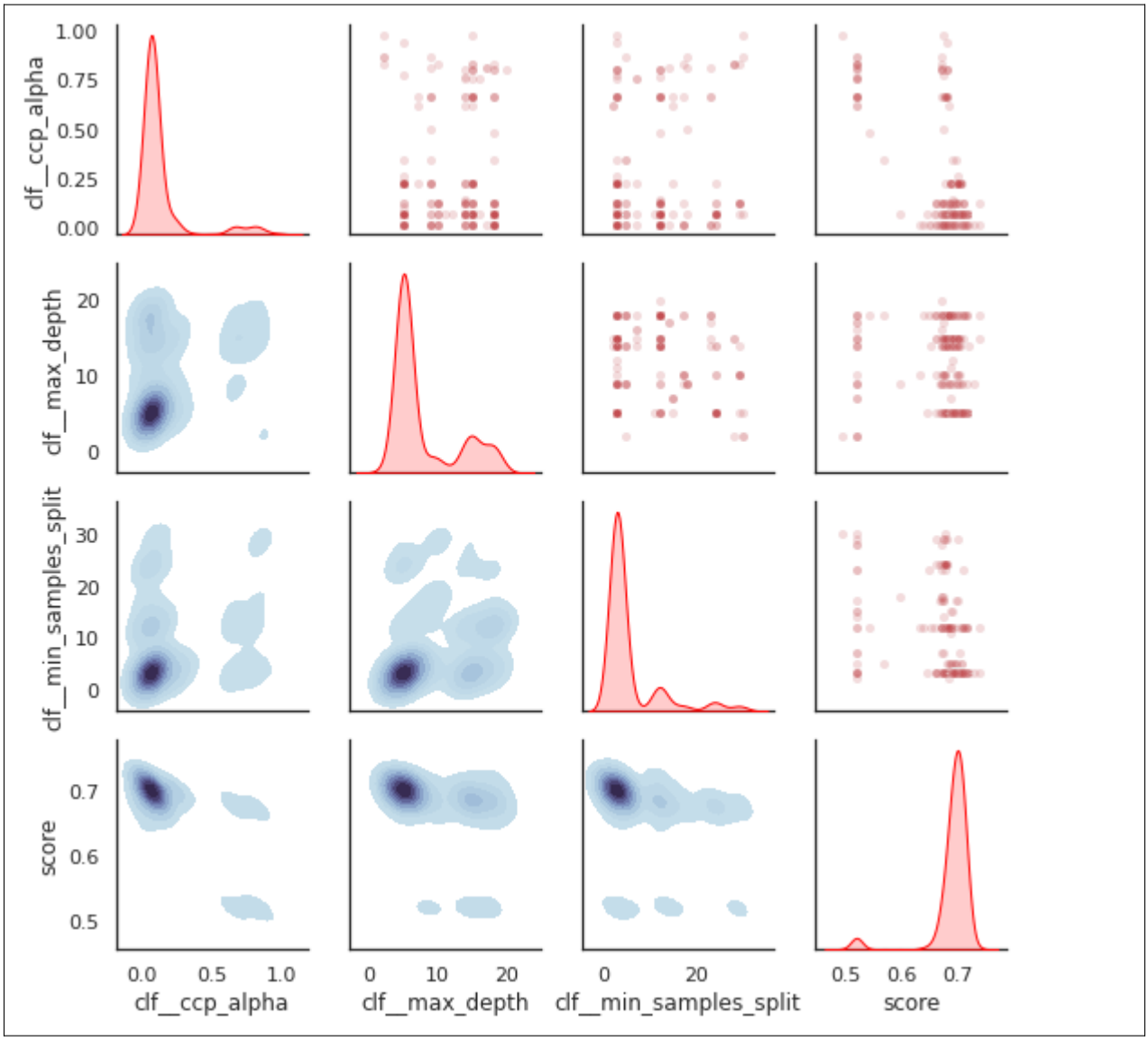
[8]:
```
print(evolved_estimator.best_params_)
print("r-squared: ", "{:.2f}".format(r_squared))
print("Best k solutions: ", evolved_estimator.hof)
```

```
{'clf__ccp_alpha': 0.014034324281064214, 'clf__criterion': 'mae', 'clf__max_depth': 18,
↪'clf__min_samples_split': 12}
r-squared:  0.67
Best k solutions:  {0: {'clf__ccp_alpha': 0.014034324281064214, 'clf__criterion': 'mae',
↪'clf__max_depth': 18, 'clf__min_samples_split': 12}, 1: {'clf__ccp_alpha': 0.
↪12139328299577712, 'clf__criterion': 'mae', 'clf__max_depth': 15, 'clf__min_samples_
↪split': 3}, 2: {'clf__ccp_alpha': 0.014034324281064214, 'clf__criterion': 'mae', 'clf__
↪max_depth': 9, 'clf__min_samples_split': 3}, 3: {'clf__ccp_alpha': 0.
↪014034324281064214, 'clf__criterion': 'mae', 'clf__max_depth': 14, 'clf__min_samples_
↪split': 3}}
```

[9]:
```
plot = plot_fitness_evolution(evolved_estimator, metric="fitness")
plt.show()
```



[10]:
```
plot_search_space(evolved_estimator)
plt.show()
```



1.10 Iris Feature Selection

[1]:
```
import matplotlib.pyplot as plt
from sklearn_genetic import GAFeatureSelectionCV
from sklearn_genetic.plots import plot_fitness_evolution
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.svm import SVC
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score
import numpy as np
```

1.10.1 Import the data and split it in train and test sets

Random noise is added to simulate useless variables

[2]:
```
data = load_iris()
X, y = data["data"], data["target"]

noise = np.random.uniform(0, 10, size=(X.shape[0], 10))

X = np.hstack((X, noise))
X.shape
```

[2]:
```
(150, 14)
```

1.10.2 Split the training and test data

[3]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=0)
```

1.10.3 Define the GAFeatureSelectionCV options

[4]:
```
clf = SVC(gamma='auto')

evolved_estimator = GAFeatureSelectionCV(
    estimator=clf,
    cv=3,
    scoring="accuracy",
    population_size=30,
    generations=20,
    n_jobs=-1,
    verbose=True,
    keep_top_k=2,
    elitism=True,
)
```

1.10.4 Fit the model and see some results

[5]:
```
evolved_estimator.fit(X, y)
features = evolved_estimator.best_features_

# Predict only with the subset of selected features
y_predict_ga = evolved_estimator.predict(X_test[:, features])
accuracy = accuracy_score(y_test, y_predict_ga)
```

```
INSTANCE
True
gen     nevals  fitness         fitness_std     fitness_max     fitness_min
0       30      0.558444        0.155441         0.893333        0.253333
1       54      0.659333        0.132948         0.893333        0.333333
2       54      0.742667        0.0867111        0.893333        0.586667
```

(continues on next page)

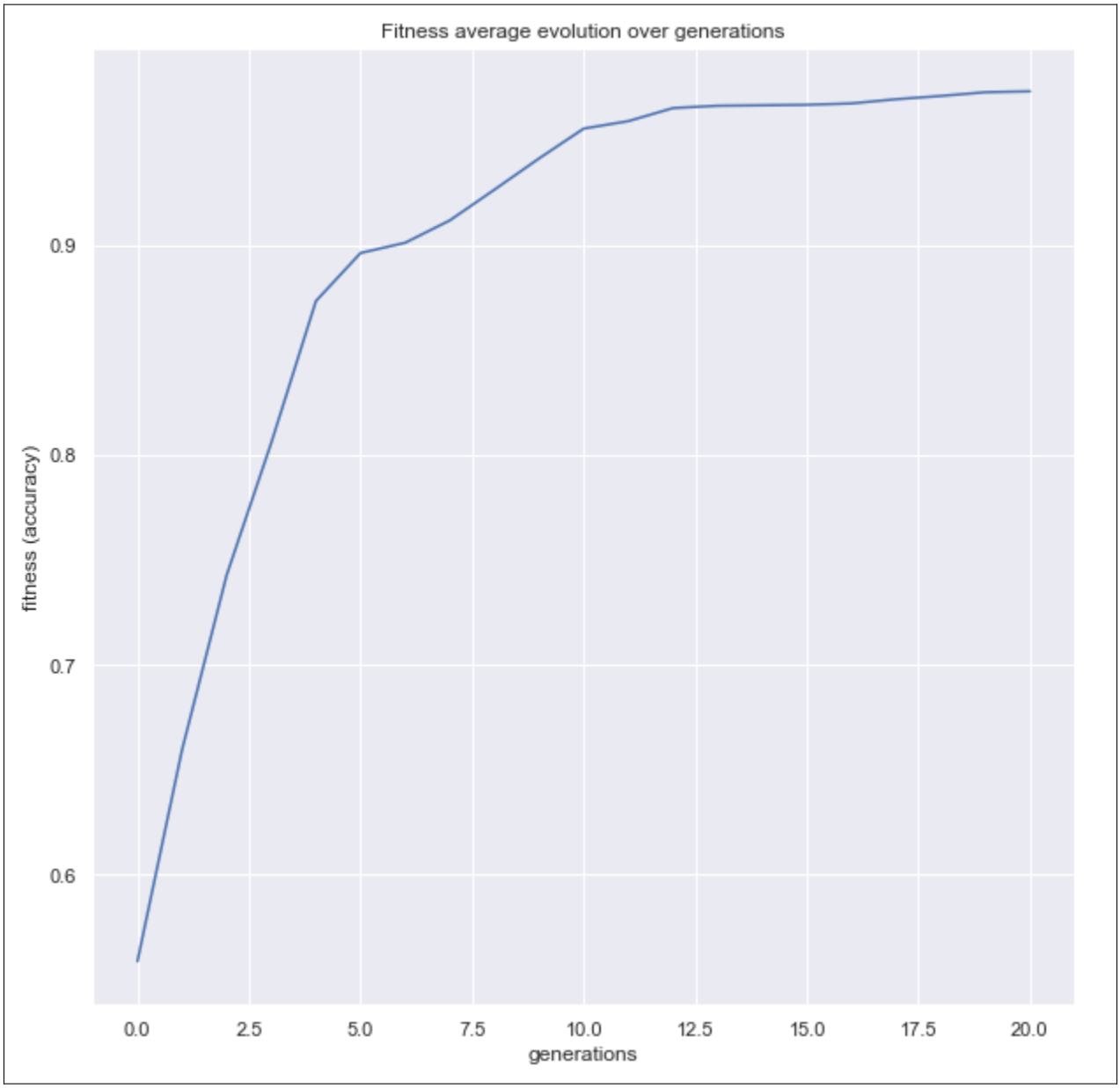
(continued from previous page)

```
3       55      0.805778        0.0740117       0.893333        0.653333
4       52      0.873333        0.0435125       0.906667        0.746667
5       53      0.896222        0.00659592      0.913333        0.893333
6       55      0.901111        0.0131186       0.953333        0.893333
7       54      0.911778        0.0206332       0.953333        0.893333
8       50      0.926444        0.0210455       0.953333        0.893333
9       51      0.941333        0.020177        0.966667        0.913333
10      49      0.955556        0.00978787      0.966667        0.913333
11      55      0.959111        0.00660714      0.966667        0.953333
12      57      0.965333        0.004           0.966667        0.953333
13      55      0.966444        0.00271257      0.973333        0.953333
14      58      0.966667        6.66134e-16     0.966667        0.966667
15      53      0.966889        0.0011967       0.973333        0.966667
16      56      0.967556        0.00226623      0.973333        0.966667
17      53      0.969556        0.00330357      0.973333        0.966667
18      51      0.971111        0.0031427       0.973333        0.966667
19      58      0.972889        0.00166296      0.973333        0.966667
20      54      0.973333        3.33067e-16     0.973333        0.973333
```

[6]:
```
print(evolved_estimator.best_features_)
print("accuracy score: ", "{:.2f}".format(accuracy))
```

```
[ True  True  True  True False False False False False False False False
 False False]
accuracy score:  0.98
```

[7]:
```
plot = plot_fitness_evolution(evolved_estimator, metric="fitness")
plt.show()
```



1.11 Digits Classification

[1]:
```
from sklearn_genetic import GASearchCV
from sklearn_genetic.space import Categorical, Integer, Continuous
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_digits
from sklearn.metrics import accuracy_score
from sklearn_genetic.callbacks import DeltaThreshold, TimerStopping
```

1.11.1 Import the data and split it in train and test sets

[2]:
```
data = load_digits()
label_names = data["target_names"]
y = data["target"]
X = data["data"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_
↪state=42)
```

1.11.2 Define the classifier to tune and the param grid

[3]:
```
clf = DecisionTreeClassifier()

params_grid = {
    "min_weight_fraction_leaf": Continuous(0, 0.5),
    "criterion": Categorical(["gini", "entropy"]),
    "max_depth": Integer(2, 20),
    "max_leaf_nodes": Integer(2, 30),
}
```

1.11.3 Create the CV strategy and optionally some callbacks

[4]:
```
cv = StratifiedKFold(n_splits=3, shuffle=True)

delta_callback = DeltaThreshold(threshold=0.001, metric="fitness")
timer_callback = TimerStopping(total_seconds=60)

callbacks = [delta_callback, timer_callback]
```

1.11.4 Define the GASearchCV options

[5]:
```
evolved_estimator = GASearchCV(
    clf,
    cv=cv,
    scoring="accuracy",
    population_size=16,
    generations=30,
    crossover_probability=0.9,
    mutation_probability=0.05,
    param_grid=params_grid,
    algorithm="eaSimple",
    n_jobs=-1,
    verbose=True)
```

1.11.5 Fit the model and see some results

[6]:
```
evolved_estimator.fit(X_train, y_train, callbacks=callbacks)
y_predict_ga = evolved_estimator.predict(X_test)
accuracy = accuracy_score(y_test, y_predict_ga)
```

```
  0%|          | 0/31 [00:00<?, ?it/s]
```

```
gen     nevals  fitness         fitness_std    fitness_max    fitness_min
0       16      0.363259        0.136399        0.639235        0.189526
1       14      0.450592        0.119266        0.620116        0.27847
2       12      0.54707         0.1376          0.75478         0.26517
3       12      0.625052        0.113433        0.768911        0.346633
4       16      0.667654        0.11493         0.755611        0.400665
5       14      0.727504        0.0156019       0.759767        0.689111
6       16      0.71462         0.0486477       0.758105        0.607648
7       14      0.701164        0.132646        0.764755        0.190357
8       12      0.735661        0.0115332       0.758936        0.715711
9       16      0.735141        0.00947264      0.748961        0.704073
INFO: DeltaThreshold callback met its criteria
INFO: Stopping the algorithm
```

[7]:
```
print(evolved_estimator.best_params_)
print("accuracy score: ", "{:.2f}".format(accuracy))
```

```
{'min_weight_fraction_leaf': 0.027793264515431237, 'criterion': 'entropy', 'max_depth':␣
↪17, 'max_leaf_nodes': 26}
accuracy score:  0.77
```

1.12 MLflow Logger

[1]:
```
from sklearn_genetic import GASearchCV
from sklearn_genetic.space import Categorical, Integer, Continuous
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_digits
from sklearn.metrics import accuracy_score
from sklearn_genetic.mlflow_log import MLflowConfig
```

1.12.1 Import the data and split it in train and test sets

[2]:
```
data = load_digits()
label_names = data["target_names"]
y = data["target"]
X = data["data"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_
↪state=42)
```

1.12.2 Define the classifier to tune and the param grid

[3]:
```
clf = DecisionTreeClassifier()

params_grid = {
    "min_weight_fraction_leaf": Continuous(0, 0.5),
    "criterion": Categorical(["gini", "entropy"]),
    "max_depth": Integer(2, 20),
    "max_leaf_nodes": Integer(2, 30)}
```

1.12.3 Create the CV strategy

[4]:
```
cv = StratifiedKFold(n_splits=3, shuffle=True)
```

1.12.4 Create the MLflowConfig object and define its options

[5]:
```
mlflow_config = MLflowConfig(
    tracking_uri="http://localhost:5000",
    experiment="Digits-sklearn-genetic-opt",
    run_name="Decision Tree",
    save_models=True,
    tags={"team": "sklearn-genetic-opt", "version": "0.5.0"})
```

```
INFO: 'Digits-sklearn-genetic-opt' does not exist. Creating a new experiment
```

1.12.5 Define the GASearchCV options

[6]:
```
evolved_estimator = GASearchCV(
    clf,
    cv=cv,
    scoring="accuracy",
    population_size=4,
    generations=10,
    crossover_probability=0.9,
    mutation_probability=0.05,
    param_grid=params_grid,
    algorithm="eaMuPlusLambda",
    n_jobs=-1,
    verbose=True,
    log_config=mlflow_config)
```

1.12.6 Fit the model and see some results

[7]:
```
evolved_estimator.fit(X_train, y_train)
y_predict_ga = evolved_estimator.predict(X_test)
accuracy = accuracy_score(y_test, y_predict_ga)
```

```
gen     nevals  fitness         fitness_std    fitness_max    fitness_min
0       4       0.261638        0.046403        0.310889        0.18537
1       8       0.32419         0.0275257       0.344971        0.276808
2       8       0.342893        0.0133196       0.353283        0.320033
3       8       0.35079         0.00249377      0.353283        0.348296
4       8       0.341854        0.013305        0.353283        0.319202
5       8       0.335619        0.0104549       0.348296        0.319202
6       7       0.339983        0.011291        0.349958        0.322527
7       7       0.354115        0.00275696      0.356608        0.349958
8       8       0.352452        0.0054509       0.356608        0.343308
9       7       0.351621        0.00498753      0.356608        0.343308
10      8       0.349543        0.00552957      0.356608        0.34414
```

[8]:
```
print(evolved_estimator.best_params_)
print("accuracy score: ", "{:.2f}".format(accuracy))
```

```
{'min_weight_fraction_leaf': 0.22010341437935194, 'criterion': 'gini', 'max_depth': 18,
↪'max_leaf_nodes': 12}
accuracy score:  0.32
```

1.13 Iris Multi-metric

[1]:
```
from sklearn_genetic import GASearchCV
from sklearn_genetic.space import Categorical, Integer, Continuous
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_iris
from sklearn.metrics import make_scorer
from sklearn.metrics import balanced_accuracy_score
```

1.13.1 Import the data and split it in train and test sets

[2]:
```
data = load_iris()
X, y = data["data"], data["target"]

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.33, random_state=0)
```

1.13.2 Define the GASearchCV options and Multi-metric

[3]:
```
clf = DecisionTreeClassifier()

params_grid = {
    "min_weight_fraction_leaf": Continuous(0, 0.5),
    "criterion": Categorical(["gini", "entropy"]),
    "max_depth": Integer(2, 20),
    "max_leaf_nodes": Integer(2, 30),
}

scoring = {"accuracy": "accuracy",
           "balanced_accuracy": make_scorer(balanced_accuracy_score)}
```

1.13.3 Define the GASearchCV options

[4]:
```
# Low number of generations and population
# Just to see the effect of multimetric
# In logbook and cv_results_

evolved_estimator = GASearchCV(
    clf,
    scoring=scoring,
    population_size=3,
    generations=2,
    crossover_probability=0.9,
    mutation_probability=0.05,
    param_grid=params_grid,
    algorithm="eaSimple",
    n_jobs=-1,
    verbose=True,
    error_score='raise',
    refit="accuracy")
```

1.13.4 Fit the model and see some results

[5]:
```
evolved_estimator.fit(X_train, y_train)
y_predict_ga = evolved_estimator.predict(X_test)
```

```
gen     nevals  fitness         fitness_std    fitness_max    fitness_min
0       3       0.856902        0.117921        0.940285        0.690137
1       2       0.940285        0               0.940285        0.940285
2       2       0.940285        0               0.940285        0.940285
```

[6]:
```
evolved_estimator.cv_results_
```

[6]:
```
{'param_min_weight_fraction_leaf': [0.22963955365985156,
  0.11807874354582698,
  0.4566955700628974,
  0.11807874354582698,
  0.22963955365985156,
  0.22963955365985156,
  0.22963955365985156],
 'param_criterion': ['gini',
  'entropy',
  'entropy',
  'gini',
  'entropy',
  'entropy',
  'gini'],
 'param_max_depth': [2, 9, 10, 2, 9, 2, 9],
 'param_max_leaf_nodes': [13, 7, 3, 7, 13, 13, 13],
 'split0_test_accuracy': [0.9117647058823529,
  0.9117647058823529,
  0.6764705882352942,
  0.9117647058823529,
  0.9117647058823529,
  0.9117647058823529,
  0.9117647058823529],
 'split1_test_accuracy': [0.9696969696969697,
  0.9696969696969697,
  0.696969696969697,
  0.9696969696969697,
  0.9696969696969697,
  0.9696969696969697,
  0.9696969696969697],
 'split2_test_accuracy': [0.9393939393939394,
  0.9393939393939394,
  0.696969696969697,
  0.9393939393939394,
  0.9393939393939394,
  0.9393939393939394,
  0.9393939393939394],
 'mean_test_accuracy': [0.9402852049910874,
  0.9402852049910874,
  0.690136660724896,
  0.9402852049910874,
```

(continues on next page)

(continued from previous page)

```
 0.9402852049910874,
 0.9402852049910874,
 0.9402852049910874],
'std_test_accuracy': [0.023659142890153965,
 0.023659142890153965,
 0.009663372529584432,
 0.023659142890153965,
 0.023659142890153965,
 0.023659142890153965,
 0.023659142890153965],
'rank_test_accuracy': array([1, 1, 7, 1, 1, 1, 1]),
'split0_train_accuracy': [0.9696969696969697,
 0.9696969696969697,
 0.696969696969697,
 0.9696969696969697,
 0.9696969696969697,
 0.9696969696969697,
 0.9696969696969697],
'split1_train_accuracy': [0.9552238805970149,
 0.9552238805970149,
 0.6865671641791045,
 0.9552238805970149,
 0.9552238805970149,
 0.9552238805970149,
 0.9552238805970149],
'split2_train_accuracy': [0.9701492537313433,
 0.9701492537313433,
 0.6865671641791045,
 0.9701492537313433,
 0.9701492537313433,
 0.9701492537313433,
 0.9701492537313433],
'mean_train_accuracy': [0.9650233680084427,
 0.9650233680084427,
 0.6900346751093019,
 0.9650233680084427,
 0.9650233680084427,
 0.9650233680084427,
 0.9650233680084427],
'std_train_accuracy': [0.006931743665052123,
 0.006931743665052123,
 0.004903800985162277,
 0.006931743665052123,
 0.006931743665052123,
 0.006931743665052123,
 0.006931743665052123],
'rank_train_accuracy': array([1, 1, 7, 1, 1, 1, 1]),
'split0_test_balanced_accuracy': [0.9090909090909092,
 0.9090909090909092,
 0.6666666666666666,
 0.9090909090909092,
 0.9090909090909092,
 0.9090909090909092,
```

(continues on next page)

(continued from previous page)

```
 0.9090909090909092,
 0.9090909090909092],
'split1_test_balanced_accuracy': [0.9722222222222222,
 0.9722222222222222,
 0.6666666666666666,
 0.9722222222222222,
 0.9722222222222222,
 0.9722222222222222,
 0.9722222222222222],
'split2_test_balanced_accuracy': [0.9333333333333332,
 0.9388888888888888,
 0.6666666666666666,
 0.9333333333333332,
 0.9388888888888888,
 0.9333333333333332,
 0.9333333333333332],
'mean_test_balanced_accuracy': [0.9382154882154882,
 0.94006734006734,
 0.6666666666666666,
 0.9382154882154882,
 0.94006734006734,
 0.9382154882154882,
 0.9382154882154882],
'std_test_balanced_accuracy': [0.02600342607735869,
 0.02578671796107406,
 0.0,
 0.02600342607735869,
 0.02578671796107406,
 0.02600342607735869,
 0.02600342607735869],
'rank_test_balanced_accuracy': array([3, 1, 7, 3, 1, 3, 3]),
'split0_train_balanced_accuracy': [0.9722222222222222,
 0.9722222222222222,
 0.6666666666666666,
 0.9722222222222222,
 0.9722222222222222,
 0.9722222222222222,
 0.9722222222222222],
'split1_train_balanced_accuracy': [0.9551414768806072,
 0.9551414768806072,
 0.6666666666666666,
 0.9551414768806072,
 0.9551414768806072,
 0.9551414768806072,
 0.9551414768806072],
'split2_train_balanced_accuracy': [0.9710144927536232,
 0.9710144927536232,
 0.6666666666666666,
 0.9710144927536232,
 0.9710144927536232,
 0.9710144927536232,
 0.9710144927536232],
```

(continues on next page)

(continued from previous page)


```
 'mean_train_balanced_accuracy': [0.9661260639521508,
  0.9661260639521508,
  0.6666666666666666,
  0.9661260639521508,
  0.9661260639521508,
  0.9661260639521508,
  0.9661260639521508],
 'std_train_balanced_accuracy': [0.007782909373174586,
  0.007782909373174586,
  0.0,
  0.007782909373174586,
  0.007782909373174586,
  0.007782909373174586,
  0.007782909373174586],
 'rank_train_balanced_accuracy': array([1, 1, 7, 1, 1, 1, 1]),
 'mean_fit_time': [0.001999060312906901,
  0.0016531944274902344,
  0.0016682147979736328,
  0.0019936561584472656,
  0.0016682942708333333,
  0.0023442904154459634,
  0.0016681353251139324],
 'std_fit_time': [8.104673248279548e-07,
  0.0004813512675446084 6,
  0.0004735620798937051,
  8.939901952387178e-06,
  0.0004726081046165265 5,
  0.0004931964528559586,
  0.0004699654688748367],
 'mean_score_time': [0.0019881725311279297,
  0.0026591618855794272,
  0.0026796658833821616,
  0.0013376077016194 66,
  0.0013335545857747395,
  0.0023202896118164062,
  0.0023340384165445 96],
 'std_score_time': [1.2906940492414977e-05,
  0.0009508785819826155,
  0.0009365762649996311,
  0.0004723497613136105 7,
  0.0004708087579728940 5,
  0.0004528267864869186,
  0.0004710938091283571 5],
 'params': [{'min_weight_fraction_leaf': 0.22963955365985156,
   'criterion': 'gini',
   'max_depth': 2,
   'max_leaf_nodes': 13},
  {'min_weight_fraction_leaf': 0.11807874354582698,
   'criterion': 'entropy',
   'max_depth': 9,
   'max_leaf_nodes': 7},
  {'min_weight_fraction_leaf': 0.4566955700628974,
```


(continues on next page)

(continued from previous page)

```
   'criterion': 'entropy',
   'max_depth': 10,
   'max_leaf_nodes': 3},
  {'min_weight_fraction_leaf': 0.11807874354582698,
   'criterion': 'gini',
   'max_depth': 2,
   'max_leaf_nodes': 7},
  {'min_weight_fraction_leaf': 0.22963955365985156,
   'criterion': 'entropy',
   'max_depth': 9,
   'max_leaf_nodes': 13},
  {'min_weight_fraction_leaf': 0.22963955365985156,
   'criterion': 'entropy',
   'max_depth': 2,
   'max_leaf_nodes': 13},
  {'min_weight_fraction_leaf': 0.22963955365985156,
   'criterion': 'gini',
   'max_depth': 9,
   'max_leaf_nodes': 13}]}
```

[7]:
```
evolved_estimator.logbook.chapters["parameters"]
```

[7]:
```
[{'index': 0,
  'min_weight_fraction_leaf': 0.22963955365985156,
  'criterion': 'gini',
  'max_depth': 2,
  'max_leaf_nodes': 13,
  'score': 0.9402852049910874,
  'cv_scores': array([0.91176471, 0.96969697, 0.93939394]),
  'fit_time': array([0.00199986, 0.00199795, 0.00199938]),
  'score_time': array([0.00197005, 0.00199533, 0.00199914]),
  'test_accuracy': array([0.91176471, 0.96969697, 0.93939394]),
  'train_accuracy': array([0.96969697, 0.95522388, 0.97014925]),
  'test_balanced_accuracy': array([0.90909091, 0.97222222, 0.93333333]),
  'train_balanced_accuracy': array([0.97222222, 0.95514148, 0.97101449])},
 {'index': 1,
  'min_weight_fraction_leaf': 0.11807874354582698,
  'criterion': 'entropy',
  'max_depth': 9,
  'max_leaf_nodes': 7,
  'score': 0.9402852049910874,
  'cv_scores': array([0.91176471, 0.96969697, 0.93939394]),
  'fit_time': array([0.00200057, 0.0019865 , 0.00097251]),
  'score_time': array([0.00200391, 0.00196981, 0.00400376]),
  'test_accuracy': array([0.91176471, 0.96969697, 0.93939394]),
  'train_accuracy': array([0.96969697, 0.95522388, 0.97014925]),
  'test_balanced_accuracy': array([0.90909091, 0.97222222, 0.93888889]),
  'train_balanced_accuracy': array([0.97222222, 0.95514148, 0.97101449])},
 {'index': 2,
  'min_weight_fraction_leaf': 0.4566955700628974,
```

(continues on next page)

(continued from previous page)

```
 'criterion': 'entropy',
 'max_depth': 10,
 'max_leaf_nodes': 3,
 'score': 0.690136660724896,
 'cv_scores': array([0.67647059, 0.6969697 , 0.6969697 ]),
 'fit_time': array([0.00200272, 0.00200343, 0.0009985 ]),
 'score_time': array([0.00203657, 0.00199842, 0.004004  ]),
 'test_accuracy': array([0.67647059, 0.6969697 , 0.6969697 ]),
 'train_accuracy': array([0.6969697 , 0.68656716, 0.68656716]),
 'test_balanced_accuracy': array([0.66666667, 0.66666667, 0.66666667]),
 'train_balanced_accuracy': array([0.66666667, 0.66666667, 0.66666667])},
{'index': 3,
 'min_weight_fraction_leaf': 0.11807874354582698,
 'criterion': 'gini',
 'max_depth': 2,
 'max_leaf_nodes': 7,
 'score': 0.9402852049910874,
 'cv_scores': array([0.91176471, 0.96969697, 0.93939394]),
 'fit_time': array([0.00200033, 0.00199962, 0.00198102]),
 'score_time': array([0.00200558, 0.00099778, 0.00100946]),
 'test_accuracy': array([0.91176471, 0.96969697, 0.93939394]),
 'train_accuracy': array([0.96969697, 0.95522388, 0.97014925]),
 'test_balanced_accuracy': array([0.90909091, 0.97222222, 0.93333333]),
 'train_balanced_accuracy': array([0.97222222, 0.95514148, 0.97101449])},
{'index': 4,
 'min_weight_fraction_leaf': 0.22963955365985156,
 'criterion': 'entropy',
 'max_depth': 9,
 'max_leaf_nodes': 13,
 'score': 0.9402852049910874,
 'cv_scores': array([0.91176471, 0.96969697, 0.93939394]),
 'fit_time': array([0.00200105, 0.00200391, 0.00099993]),
 'score_time': array([0.00100136, 0.00099993, 0.00199938]),
 'test_accuracy': array([0.91176471, 0.96969697, 0.93939394]),
 'train_accuracy': array([0.96969697, 0.95522388, 0.97014925]),
 'test_balanced_accuracy': array([0.90909091, 0.97222222, 0.93888889]),
 'train_balanced_accuracy': array([0.97222222, 0.95514148, 0.97101449])},
{'index': 5,
 'min_weight_fraction_leaf': 0.22963955365985156,
 'criterion': 'entropy',
 'max_depth': 2,
 'max_leaf_nodes': 13,
 'score': 0.9402852049910874,
 'cv_scores': array([0.91176471, 0.96969697, 0.93939394]),
 'fit_time': array([0.00304174, 0.00198984, 0.00200129]),
 'score_time': array([0.00296068, 0.00200129, 0.0019989 ]),
 'test_accuracy': array([0.91176471, 0.96969697, 0.93939394]),
 'train_accuracy': array([0.96969697, 0.95522388, 0.97014925]),
 'test_balanced_accuracy': array([0.90909091, 0.97222222, 0.93333333]),
 'train_balanced_accuracy': array([0.97222222, 0.95514148, 0.97101449])},
{'index': 6,
 'min_weight_fraction_leaf': 0.22963955365985156,
```

(continues on next page)

(continued from previous page)

```
 'criterion': 'gini',
 'max_depth': 9,
 'max_leaf_nodes': 13,
 'score': 0.9402852049910874,
 'cv_scores': array([0.91176471, 0.96969697, 0.93939394]),
 'fit_time': array([0.00200057, 0.0010035 , 0.00200033]),
 'score_time': array([0.00200343, 0.00300026, 0.00199842]),
 'test_accuracy': array([0.91176471, 0.96969697, 0.93939394]),
 'train_accuracy': array([0.96969697, 0.95522388, 0.97014925]),
 'test_balanced_accuracy': array([0.90909091, 0.97222222, 0.93333333]),
 'train_balanced_accuracy': array([0.97222222, 0.95514148, 0.97101449])}]
```

1.14 Release Notes

Some notes on new features in various releases

1.14.1 What's new in 0.9.0

Features:

- Introducing Adaptive Schedulers to enable adaptive mutation and crossover probabilities; currently, supported schedulers are:
  - *ConstantAdapter*
  - *ExponentialAdapter*
  - *InverseAdapter*
  - *PotentialAdapter*
- Add *random_state* parameter (default= `None`) in *Continuous*, *Categorical* and *Integer* classes to leave fixed the random seed during hyperparameters sampling. Take into account that this only ensures that the space components are reproducible, not all the package. This is due to the DEAP dependency, which doesn't seem to have a native way to set the random seed.

API Changes:

- Changed the default values of *mutation_probability* and *crossover_probability* to 0.8 and 0.2, respectively.
- The *weighted_choice* function used in *GAFeatureSelectionCV* was re-written to give more probability to a number of features closer to the *max_features* parameter
- Removed unused and wrong function `plot_parallel_coordinates()`

Bug Fixes:

- Now when using the *plot_search_space()* function, all the parameters get casted as np.float64 to avoid errors on seaborn package while plotting bool values.

1.14.2 What's new in 0.8.1

Features:

- If the *max_features* parameter from *GAFeatureSelectionCV* is set, the initial population is now sampled giving more probability to solutions with less than *max_features* features.

1.14.3 What's new in 0.8.0

Features:

- *GAFeatureSelectionCV* now has a parameter called *max_features*, int, default=None. If it's not None, it will penalize individuals with more features than max_features, putting a "soft" upper bound to the number of features to be selected.
- Classes *GASearchCV* and *GAFeatureSelectionCV* now support multi-metric evaluation the same way scikit-learn does, you will see this reflected on the *logbook* and *cv_results_* objects, where now you get results for each metric. As in scikit-learn, if multi-metric is used, the *refit* parameter must be a str specifying the metric to evaluate the cv-scores. See more in the *GASearchCV* and *GAFeatureSelectionCV* API documentation.
- Training gracefully stops if interrupted by some of these exceptions: `KeyboardInterrupt`, `SystemExit`, `StopIteration`. When one of these exceptions is raised, the model finishes the current generation and saves the current best model. It only works if at least one generation has been completed.

API Changes:

- The following parameters changed their default values to create more extensive and different models with better results:
  - population_size from 10 to 50
  - generations from 40 to 80
  - mutation_probability from 0.1 to 0.2

Docs:

- A new notebook called Iris_multimetric was added to showcase the new multi-metric capabilities.

1.14.4 What's new in 0.7.0

Features:

- *GAFeatureSelectionCV* for feature selection along with any scikit-learn classifier or regressor. It optimizes the cv-score while minimizing the number of features to select. This class is compatible with the mlflow and tensorboard integration, the Callbacks and the `plot_fitness_evolution` function.

API Changes:

- The module `mlflow` was renamed to `mlflow_log` to avoid unexpected errors on name resolutions

1.14.5 What's new in 0.6.1

Features:

- Added the parameter *generations* to the *DeltaThreshold*. Now it compares the maximum and minimum values of a metric from the last generations, instead of just the current and previous ones. The default value is 2, so the behavior remains the same as in previous versions.

Bug Fixes:

- When a param_grid of length 1 is provided, a user warning is raised instead of an error. Internally it will swap the crossover operation to use the DEAP's `cxSimulatedBinaryBounded()`.
- When using *Continuous* class with boundaries *lower* and *upper*, a uniform distribution with limits *[lower, lower + upper]* was sampled, now, it's properly sampled using a *[lower, upper]* limits.

1.14.6 What's new in 0.6.0

Features:

- Added the *ProgressBar* callback, it uses tqdm progress bar to shows how many generations are left in the training progress.
- Added the *TensorBoard* callback to log the generation metrics, watch in real time while the models are trained and compare different runs in your TensorBoard instance.
- Added the *TimerStopping* callback to stop the iterations after a total (threshold) fitting time has been elapsed.
- Added new parallel coordinates plot in `plot_parallel_coordinates()`.
- Now if one or more callbacks decides to stop the algorithm, it will print its class name to know which callbacks were responsible of the stopping.
- Added support for extra methods coming from scikit-learn's BaseSearchCV, like *cv_results_*, *best_index_* and *refit_time_* among others.
- Added methods *on_start* and *on_end* to *BaseCallback*. Now the algorithms check for the callbacks like this:
  - **on_start**: When the evolutionary algorithm is called from the GASearchCV.fit method.
  - **on_step:** When the evolutionary algorithm finishes a generation (no change here).
  - **on_end:** At the end of the last generation.

Bug Fixes:

- A missing statement was making that the callbacks start to get evaluated from generation 1, ignoring generation 0. Now this is properly handled and callbacks work from generation 0.

API Changes:

- The modules *plots* and `MLflowConfig` now requires an explicit installation of seaborn and mlflow, now those are optionally installed using `pip install sklearn-genetic-opt[all]`.
- The GASearchCV.logbook property now has extra information that comes from the scikit-learn cross_validate function.
- An optional extra parameter was added to GASearchCV, named *return_train_score*: bool, default= `False`. As in scikit-learn, it controls if the *cv_results_* should have the training scores.

Docs:

- Edited all demos to be in the jupyter notebook format.
- Added embedded jupyter notebooks examples.
- The modules of the package now have a summary of their classes/functions in the docs.
- Updated the callbacks and custom callbacks tutorials to add new TensorBoard callback and the new methods on the base callback.

Internal:

- Now the hof uses the *self.best_params_* for the position 0, to be consistent with the scikit-learn API and parameters like *self.best_index_*

1.14.7 What's new in 0.5.0

Features:

- Build-in integration with MLflow using the `MLflowConfig` and the new parameter *log_config* from *GASearchCV*
- Implemented the callback *LogbookSaver* which saves the estimator.logbook object with all the fitted hyperparameters and their cross-validation score
- Added the parameter *estimator* to all the functions on the module `callbacks`

Docs:

- Added user guide "Integrating with MLflow"
- Update the tutorial "Custom Callbacks" for new API inheritance behavior

Internal:

- Added a base class `BaseCallback` from which all Callbacks must inherit from
- Now coverage report doesn't take into account the lines with # pragma: no cover and # noqa

1.14.8 What's new in 0.4.1

Docs:

- Added user guide on "Understanding the evaluation process"
- Several guides on contributing, code of conduct
- Added important links
- Docs requirements are now independent of package requirements

Internal:

- Changed test ci from travis to Github actions

1.14.9 What's new in 0.4

Features:

- Implemented callbacks module to stop the optimization process based in the current iteration metrics, currently implemented: `ThresholdStopping` , `ConsecutiveStopping` and `DeltaThreshold`.
- The algorithms 'eaSimple', 'eaMuPlusLambda', 'eaMuCommaLambda' are now implemented in the module `algorithms` for more control over their options, rather that taking the deap.algorithms module
- Implemented the `plots` module and added the function `plot_search_space()`, this function plots a mixed counter, scatter and histogram plots over all the fitted hyperparameters and their cross-validation score
- Documentation based in rst with Sphinx to host in read the docs. It includes public classes and functions documentation as well as several tutorials on how to use the package
- Added *best_params_* and *best_estimator_* properties after fitting GASearchCV
- Added optional parameters *refit*, *pre_dispatch* and *error_score*

API Changes:

- Removed support for python 3.6, changed the libraries supported versions to be the same as scikit-learn current version
- Several internal changes on the documentation and variables naming style to be compatible with Sphinx
- Removed the parameters *continuous_parameters*, *categorical_parameters* and *integer_parameters* replacing them with *param_grid*

1.14.10 What's new in 0.3

Features:

- Added the space module to control better the data types and ranges of each hyperparameter, their distribution to sample random values from, and merge all data types in one Space class that can work with the new param_grid parameter
- Changed the *continuous_parameters*, *categorical_parameters* and *integer_parameters* for the *param_grid*, the first ones still work but will be removed in a next version
- Added the option to use the eaMuCommaLambda algorithm from deap
- The *mu* and *lambda_* parameters of the internal eaMuPlusLambda and eaMuCommaLambda now are in terms of the initial population size and not the number of generations

1.14.11 What's new in 0.2

Features:

- Enabled deap's eaMuPlusLambda algorithm for the optimization process, now is the default routine
- Added a logbook and history properties to the fitted GASearchCV to make post-fit analysis
- `Elitism=False` now implements a roulette selection instead of ignoring the parameter
- Added the parameter keep_top_k to control the number of solutions if the hall of fame (hof)

API Changes:

- Refactored the optimization algorithm to use DEAP package instead of a custom implementation, this causes the removal of several methods, properties and variables inside the GASearchCV class
- The parameter encoding_length has been removed, it's no longer required to the GASearchCV class
- Renamed the property of the fitted estimator from *best_params_* to *best_params*
- The verbosity now prints the deap log of the fitness function, it's standard deviation, max and min values from each generation
- The variable *GASearchCV._best_solutions* was removed and it's meant to be replaced with *GASearchCV.logbook* and *GASearchCV.history*
- Changed default parameters crossover_probability from 1 to 0.8 and generations from 50 to 40

1.14.12 What's new in 0.1

Features:

- *GASearchCV* for hyperparameters tuning using custom genetic algorithm for scikit-learn classification and regression models
- *plot_fitness_evolution()* function to see the average fitness values over generations

1.15 GASearchCV

GASearchCV(estimator[, cv, param_grid, ...])	Evolutionary optimization over hyperparameters.
GASearchCV.decision_function(X)	Call decision_function on the estimator with the best found parameters.
GASearchCV.fit(X, y[, callbacks])	Main method of GASearchCV, starts the optimization procedure with the hyperparameters of the given estimator
GASearchCV.get_params([deep])	Get parameters for this estimator.
GASearchCV.inverse_transform(Xt)	Call inverse_transform on the estimator with the best found params.
GASearchCV.predict(X)	Call predict on the estimator with the best found parameters.
GASearchCV.predict_proba(X)	Call predict_proba on the estimator with the best found parameters.
GASearchCV.score(X[, y])	Return the score on the given data, if the estimator has been refit.
GASearchCV.score_samples(X)	Call score_samples on the estimator with the best found parameters.
GASearchCV.set_params(**params)	Set the parameters of this estimator.
GASearchCV.transform(X)	Call transform on the estimator with the best found parameters.

class sklearn_genetic.**GASearchCV**(*estimator*, *cv=3*, *param_grid=None*, *scoring=None*, *population_size=50*, *generations=80*, *crossover_probability=0.2*, *mutation_probability=0.8*, *tournament_size=3*, *elitism=True*, *verbose=True*, *keep_top_k=1*, *criteria='max'*, *algorithm='eaMuPlusLambda'*, *refit=True*, *n_jobs=1*, *pre_dispatch='2*n_jobs'*, *error_score=nan*, *return_train_score=False*, *log_config=None*)

Evolutionary optimization over hyperparameters.

GASearchCV implements a "fit" and a "score" method. It also implements "predict", "predict_proba", "decision_function", "predict_log_proba" if they are implemented in the estimator used. The parameters of the estimator used to apply these methods are optimized by cross-validated search over parameter settings.

Parameters

estimator
[estimator object, default=None] estimator object implementing 'fit' The object to use to fit the data.

cv
[int, cross-validation generator or an iterable, default=None] Determines the cross-validation splitting strategy. Possible inputs for cv are: - None, to use the default 5-fold cross validation, - int, to specify the number of folds in a *(Stratified)KFold*, - CV splitter, - An iterable yielding (train, test) splits as arrays of indices. For int/None inputs, if the estimator is a classifier and y is either binary or multiclass, `StratifiedKFold` is used. In all other cases, `KFold` is used. These splitters are instantiated with *shuffle=False* so the splits will be the same across calls.

param_grid
[dict, default=None] Grid with the parameters to tune, expects keys a valid name of hyperparameter based on the estimator selected and as values one of *Integer* , *Categorical* *Continuous* classes. At least two parameters are advised to be provided in order to successfully make an optimization routine.

population_size
[int, default=10] Size of the initial population to sample randomly generated individuals.

generations
[int, default=40] Number of generations or iterations to run the evolutionary algorithm.

crossover_probability
[float or a Scheduler, default=0.8] Probability of crossover operation between two individuals.

mutation_probability
[float or a Scheduler, default=0.1] Probability of child mutation.

tournament_size
[int, default=3] Number of individuals to perform tournament selection.

elitism
[bool, default=True] If True takes the *tournament_size* best solution to the next generation.

scoring
[str, callable, list, tuple or dict, default=None] Strategy to evaluate the performance of the cross-validated model on the test set. If *scoring* represents a single score, one can use:

- a single string;
- a callable that returns a single value.

If *scoring* represents multiple scores, one can use:

- a list or tuple of unique strings;
- a callable returning a dictionary where the keys are the metric names and the values are the metric scores;
- a dictionary with metric names as keys and callables a values.

n_jobs
[int, default=None] Number of jobs to run in parallel. Training the estimator and computing the score are parallelized over the cross-validation splits. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors.

verbose
[bool, default=True] If `True`, shows the metrics on the optimization routine.

keep_top_k
[int, default=1] Number of best solutions to keep in the hof object. If a callback stops the algorithm before k iterations, it will return only one set of parameters per iteration.

criteria
[{'max', 'min'} , default='max'] `max` if a higher scoring metric is better, `min` otherwise.

algorithm
[{'eaMuPlusLambda', 'eaMuCommaLambda', 'eaSimple'}, default='eaMuPlusLambda'] Evolutionary algorithm to use. See more details in the deap algorithms documentation.

refit
[bool, str, or callable, default=True] Refit an estimator using the best found parameters on the whole dataset. For multiple metric evaluation, this needs to be a *str* denoting the scorer that would be used to find the best parameters for refitting the estimator at the end. The refitted estimator is made available at the `best_estimator_` attribute and permits using `predict` directly on this `GASearchCV` instance. Also for multiple metric evaluation, the attributes `best_index_`, `best_score_` and `best_params_` will only be available if `refit` is set and

all of them will be determined w.r.t this specific scorer. See `scoring` parameter to know more about multiple metric evaluation.

If `False`, it is not possible to make predictions using this GASearchCV instance after fitting.

pre_dispatch
[int or str, default='2*n_jobs'] Controls the number of jobs that get dispatched during parallel execution. Reducing this number can be useful to avoid an explosion of memory consumption when more jobs get dispatched than CPUs can process. This parameter can be:

- None, in which case all the jobs are immediately created and spawned. Use this for lightweight and fast-running jobs, to avoid delays due to on-demand spawning of the jobs
- An int, giving the exact number of total jobs that are spawned
- A str, giving an expression as a function of n_jobs, as in '2*n_jobs'

error_score
['raise' or numeric, default=np.nan] Value to assign to the score if an error occurs in estimator fitting. If set to `'raise'`, the error is raised. If a numeric value is given, FitFailedWarning is raised.

return_train_score: bool, default=False
If `False`, the `cv_results_` attribute will not include training scores. Computing training scores is used to get insights on how different parameter settings impact the overfitting/underfitting trade-off. However computing the scores on the training set can be computationally expensive and is not strictly required to select the parameters that yield the best generalization performance.

log_config
[`MLflowConfig`, default = None] Configuration to log metrics and models to mlflow, of None, no mlflow logging will be performed

Attributes

logbook
[`DEAP.tools.Logbook`] Contains the logs of every set of hyperparameters fitted with its average scoring metric.

history
[dict] Dictionary of the form: {"gen": [], "fitness": [], "fitness_std": [], "fitness_max": [], "fitness_min": []}

gen returns the index of the evaluated generations. Each entry on the others lists, represent the average metric in each generation.

cv_results_
[dict of numpy (masked) ndarrays] A dict with keys as column headers and values as columns, that can be imported into a pandas `DataFrame`.

best_estimator_
[estimator] Estimator that was chosen by the search, i.e. estimator which gave highest score on the left out data. Not available if `refit=False`.

best_params_
[dict] Parameter setting that gave the best results on the hold out data.

best_index_
[int] The index (of the `cv_results_` arrays) which corresponds to the best candidate parameter setting. The dict at `search.cv_results_['params'][search.best_index_]` gives the parameter setting for the best model, that gives the highest mean score (`search.best_score_`).

scorer_
[function or a dict] Scorer function used on the held out data to choose the best parameters for the model.

n_splits_
[int] The number of cross-validation splits (folds/iterations).

refit_time_
[float] Seconds used for refitting the best model on the whole dataset. This is present only if `refit` is not False.

`decision_function`(*X*)

Call decision_function on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `decision_function`.

Parameters

X
[indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

Returns

y_score
[ndarray of shape (n_samples,) or (n_samples, n_classes) or (n_samples, n_classes * (n_classes-1) / 2)] Result of the decision function for *X* based on the estimator with the best found parameters.

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

Main method of GASearchCV, starts the optimization procedure with the hyperparameters of the given estimator

Parameters

X
[array-like of shape (n_samples, n_features)] The data to fit. Can be for example a list, or an array.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] The target variable to try to predict in the case of supervised learning.

callbacks: list or callable
One or a list of the callbacks methods available in `callbacks`. The callback is evaluated after fitting the estimators from the generation 1.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`inverse_transform`(*Xt*)

Call inverse_transform on the estimator with the best found params.

Only available if the underlying estimator implements `inverse_transform` and `refit=True`.

Parameters

Xt
[indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

Returns

X
[{ndarray, sparse matrix} of shape (n_samples, n_features)] Result of the *inverse_transform* function for *Xt* based on the estimator with the best found parameters.

`predict`(*X*)

Call predict on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict`.

Parameters

X
[indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

Returns

y_pred
[ndarray of shape (n_samples,)] The predicted labels or values for *X* based on the estimator with the best found parameters.

`predict_log_proba`(*X*)

Call predict_log_proba on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict_log_proba`.

Parameters

X
[indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

Returns

y_pred
[ndarray of shape (n_samples,) or (n_samples, n_classes)] Predicted class log-probabilities for *X* based on the estimator with the best found parameters. The order of the classes corresponds to that in the fitted attribute classes_.

`predict_proba`(*X*)

Call predict_proba on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict_proba`.

Parameters

X
[indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

Returns

y_pred
[ndarray of shape (n_samples,) or (n_samples, n_classes)] Predicted class probabilities for *X* based on the estimator with the best found parameters. The order of the classes corresponds to that in the fitted attribute classes_.

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

Return the score on the given data, if the estimator has been refit.

This uses the score defined by `scoring` where provided, and the `best_estimator_.score` method otherwise.

Parameters

X
[array-like of shape (n_samples, n_features)] Input data, where *n_samples* is the number of samples and *n_features* is the number of features.

y
[array-like of shape (n_samples, n_output) or (n_samples,), default=None] Target relative to X for classification or regression; None for unsupervised learning.

Returns

score
[float] The score defined by `scoring` if provided, and the `best_estimator_.score` method otherwise.

`score_samples`(*X*)

Call score_samples on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `score_samples`.

New in version 0.24.

Parameters

X
[iterable] Data to predict on. Must fulfill input requirements of the underlying estimator.

Returns

y_score
[ndarray of shape (n_samples,)] The `best_estimator_.score_samples` method.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

transform(*X*)

Call transform on the estimator with the best found parameters.

Only available if the underlying estimator supports `transform` and `refit=True`.

Parameters

X

[indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

Returns

Xt

[{ndarray, sparse matrix} of shape (n_samples, n_features)] *X* transformed in the new space based on the estimator with the best found parameters.

1.16 FeatureSelectionCV

GAFeatureSelectionCV(estimator[, cv, ...])	Evolutionary optimization for feature selection.
GASearchCV.decision_function(X)	Call decision_function on the estimator with the best found parameters.
GASearchCV.fit(X, y[, callbacks])	Main method of GASearchCV, starts the optimization procedure with the hyperparameters of the given estimator
GASearchCV.get_params([deep])	Get parameters for this estimator.
GASearchCV.inverse_transform(Xt)	Call inverse_transform on the estimator with the best found params.
GASearchCV.predict(X)	Call predict on the estimator with the best found parameters.
GASearchCV.predict_proba(X)	Call predict_proba on the estimator with the best found parameters.
GASearchCV.score(X[, y])	Return the score on the given data, if the estimator has been refit.
GASearchCV.score_samples(X)	Call score_samples on the estimator with the best found parameters.
GASearchCV.set_params(**params)	Set the parameters of this estimator.
GASearchCV.transform(X)	Call transform on the estimator with the best found parameters.

class sklearn_genetic.**GAFeatureSelectionCV**(*estimator*, *cv=3*, *scoring=None*, *population_size=50*, *generations=80*, *crossover_probability=0.2*, *mutation_probability=0.8*, *tournament_size=3*, *elitism=True*, *max_features=None*, *verbose=True*, *keep_top_k=1*, *criteria='max'*, *algorithm='eaMuPlusLambda'*, *refit=True*, *n_jobs=1*, *pre_dispatch='2*n_jobs'*, *error_score=nan*, *return_train_score=False*, *log_config=None*)

Evolutionary optimization for feature selection.

GAFeatureSelectionCV implements a "fit" and a "score" method. It also implements "predict", "predict_proba", "decision_function", "predict_log_proba" if they are implemented in the estimator used. The features (variables) used by the estimator are found by optimizing the cv-scores and by minimizing the number of features

Parameters

estimator
[estimator object, default=None] estimator object implementing 'fit' The object to use to fit the data.

cv
[int, cross-validation generator or an iterable, default=None] Determines the cross-validation splitting strategy. Possible inputs for cv are: - None, to use the default 5-fold cross validation, - int, to specify the number of folds in a *(Stratified)KFold*, - CV splitter, - An iterable yielding (train, test) splits as arrays of indices. For int/None inputs, if the estimator is a classifier and `y` is either binary or multiclass, `StratifiedKFold` is used. In all other cases, `KFold` is used. These splitters are instantiated with *shuffle=False* so the splits will be the same across calls.

population_size
[int, default=10] Size of the initial population to sample randomly generated individuals.

generations
[int, default=40] Number of generations or iterations to run the evolutionary algorithm.

crossover_probability
[float or a Scheduler, default=0.2] Probability of crossover operation between two individuals.

mutation_probability
[float or a Scheduler, default=0.8] Probability of child mutation.

tournament_size
[int, default=3] Number of individuals to perform tournament selection.

elitism
[bool, default=True] If True takes the *tournament_size* best solution to the next generation.

max_features
[int, default=None] The upper bound number of features to be selected.

scoring
[str, callable, list, tuple or dict, default=None] Strategy to evaluate the performance of the cross-validated model on the test set. If *scoring* represents a single score, one can use:

- a single string;
- a callable that returns a single value.

If *scoring* represents multiple scores, one can use:

- a list or tuple of unique strings;
- a callable returning a dictionary where the keys are the metric names and the values are the metric scores;
- a dictionary with metric names as keys and callables a values.

n_jobs
[int, default=None] Number of jobs to run in parallel. Training the estimator and computing the score are parallelized over the cross-validation splits. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors.

verbose
[bool, default=True] If `True`, shows the metrics on the optimization routine.

keep_top_k
[int, default=1] Number of best solutions to keep in the hof object. If a callback stops the algorithm before k iterations, it will return only one set of parameters per iteration.

criteria
[{'max', 'min'} , default='max'] `max` if a higher scoring metric is better, `min` otherwise.

algorithm
[{'eaMuPlusLambda', 'eaMuCommaLambda', 'eaSimple'}, default='eaMuPlusLambda'] Evolutionary algorithm to use. See more details in the deap algorithms documentation.

refit
[bool, str, or callable, default=True] Refit an estimator using the best found parameters on the whole dataset. For multiple metric evaluation, this needs to be a *str* denoting the scorer that would be used to find the best parameters for refitting the estimator at the end. The refitted estimator is made available at the `best_estimator_` attribute and permits using `predict` directly on this `FeatureSelectionCV` instance. Also for multiple metric evaluation, the attributes `best_index_`, `best_score_` and `best_params_` will only be available if `refit` is set and all of them will be determined w.r.t this specific scorer. See `scoring` parameter to know more about multiple metric evaluation.

If `False`, it is not possible to make predictions using this GASearchCV instance after fitting.

pre_dispatch
[int or str, default='2*n_jobs'] Controls the number of jobs that get dispatched during parallel execution. Reducing this number can be useful to avoid an explosion of memory consumption when more jobs get dispatched than CPUs can process. This parameter can be:

- None, in which case all the jobs are immediately created and spawned. Use this for lightweight and fast-running jobs, to avoid delays due to on-demand spawning of the jobs
- An int, giving the exact number of total jobs that are spawned
- A str, giving an expression as a function of n_jobs, as in '2*n_jobs'

error_score
['raise' or numeric, default=np.nan] Value to assign to the score if an error occurs in estimator fitting. If set to `'raise'`, the error is raised. If a numeric value is given, FitFailedWarning is raised.

return_train_score: bool, default=False
If `False`, the `cv_results_` attribute will not include training scores. Computing training scores is used to get insights on how different parameter settings impact the overfitting/underfitting trade-off. However computing the scores on the training set can be computationally expensive and is not strictly required to select the parameters that yield the best generalization performance.

log_config
[`MLflowConfig`, default = None] Configuration to log metrics and models to mlflow, of None, no mlflow logging will be performed

Attributes

logbook
[`DEAP.tools.Logbook`] Contains the logs of every set of hyperparameters fitted with its average scoring metric.

history
[dict] Dictionary of the form: {"gen": [], "fitness": [], "fitness_std": [], "fitness_max": [], "fitness_min": []}

gen returns the index of the evaluated generations. Each entry on the others lists, represent the average metric in each generation.

cv_results_
[dict of numpy (masked) ndarrays] A dict with keys as column headers and values as columns, that can be imported into a pandas `DataFrame`.

best_estimator_
[estimator] Estimator that was chosen by the search, i.e. estimator which gave highest score on the left out data. Not available if `refit=False`.

best_features_
[list] List of bool, each index represents one feature in the same order the data was fed. 1 means the feature was selected, 0 means the features was discarded.

scorer_
[function or a dict] Scorer function used on the held out data to choose the best parameters for the model.

n_splits_
[int] The number of cross-validation splits (folds/iterations).

refit_time_
[float] Seconds used for refitting the best model on the whole dataset. This is present only if `refit` is not False.

`decision_function`(*X*)

Call decision_function on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `decision_function`.

Parameters

X
[indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

Returns

y_score
[ndarray of shape (n_samples,) or (n_samples, n_classes) or (n_samples, n_classes * (n_classes-1) / 2)] Result of the decision function for *X* based on the estimator with the best found parameters.

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

Main method of GAFeatureSelectionCV, starts the optimization procedure with to find the best features set Parameters ———- X : array-like of shape (n_samples, n_features)

The data to fit. Can be for example a list, or an array.

y
[array-like of shape (n_samples,) or (n_samples, n_outputs), default=None] The target variable to try to predict in the case of supervised learning.

callbacks: list or callable
One or a list of the callbacks methods available in `callbacks`. The callback is evaluated after fitting the estimators from the generation 1.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep
[bool, default=True] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params
[dict] Parameter names mapped to their values.

`inverse_transform`(*Xt*)

Call inverse_transform on the estimator with the best found params.

Only available if the underlying estimator implements `inverse_transform` and `refit=True`.

Parameters

Xt
[indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

Returns

X
[{ndarray, sparse matrix} of shape (n_samples, n_features)] Result of the *inverse_transform* function for *Xt* based on the estimator with the best found parameters.

`predict`(*X*)

Call predict on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict`.

Parameters

X
[indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

Returns

y_pred
[ndarray of shape (n_samples,)] The predicted labels or values for *X* based on the estimator with the best found parameters.

`predict_log_proba`(*X*)

Call predict_log_proba on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict_log_proba`.

Parameters

X
[indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

Returns

y_pred
[ndarray of shape (n_samples,) or (n_samples, n_classes)] Predicted class log-probabilities for *X* based on the estimator with the best found parameters. The order of the classes corresponds to that in the fitted attribute classes_.

`predict_proba`(*X*)

Call predict_proba on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `predict_proba`.

Parameters

X
[indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

Returns

y_pred
[ndarray of shape (n_samples,) or (n_samples, n_classes)] Predicted class probabilities for *X* based on the estimator with the best found parameters. The order of the classes corresponds to that in the fitted attribute classes_.

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

Return the score on the given data, if the estimator has been refit.

This uses the score defined by `scoring` where provided, and the `best_estimator_.score` method otherwise.

Parameters

X
[array-like of shape (n_samples, n_features)] Input data, where *n_samples* is the number of samples and *n_features* is the number of features.

y
[array-like of shape (n_samples, n_output) or (n_samples,), default=None] Target relative to X for classification or regression; None for unsupervised learning.

Returns

score
[float] The score defined by `scoring` if provided, and the `best_estimator_.score` method otherwise.

`score_samples`(*X*)

Call score_samples on the estimator with the best found parameters.

Only available if `refit=True` and the underlying estimator supports `score_samples`.

New in version 0.24.

Parameters

X
[iterable] Data to predict on. Must fulfill input requirements of the underlying estimator.

Returns

y_score
[ndarray of shape (n_samples,)] The `best_estimator_.score_samples` method.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as `Pipeline`). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Parameters

****params**
[dict] Estimator parameters.

Returns

self
[estimator instance] Estimator instance.

`transform`(*X*)

Call transform on the estimator with the best found parameters.

Only available if the underlying estimator supports `transform` and `refit=True`.

Parameters

X
[indexable, length n_samples] Must fulfill the input assumptions of the underlying estimator.

Returns

Xt
[{ndarray, sparse matrix} of shape (n_samples, n_features)] *X* transformed in the new space based on the estimator with the best found parameters.

1.17 Callbacks

base.BaseCallback()	Base Callback from which all Callbacks must inherit from
ProgressBar(**kwargs)	Displays a tqdm progress bar with the training progress.
ConsecutiveStopping(generations[, metric])	Stop the optimization if the current metric value is no greater that at least one metric from the last N generations
DeltaThreshold(threshold[, generations, metric])	Stops the optimization if the absolute difference between the maximum and minimum value from the last N generations is less or equals to a threshold.
TimerStopping(total_seconds)	Stops the optimization process if a limit training time has been elapsed.
ThresholdStopping(threshold[, metric])	Stop the optimization if the metric from cross validation score is greater or equals than the define threshold
TensorBoard([log_dir, run_id])	Log all the fitness metrics to Tensorboard into log_dir/run_id folder
LogbookSaver(checkpoint_path, **dump_options)	Saves the estimator.logbook parameter chapter object in a local file system.

class sklearn_genetic.callbacks.base.**BaseCallback**

Base Callback from which all Callbacks must inherit from

on_end(*logbook=None*, *estimator=None*)

Take actions at the end of the training

Parameters

logbook:
Current stream logbook with the stats required

estimator:
GASearchCV Estimator that is being optimized

`on_start`(*estimator=None*)

Take actions at the start of the training

Parameters

estimator:
GASearchCV Estimator that is being optimized

`on_step`(*record=None*, *logbook=None*, *estimator=None*)

Take actions after fitting each generation.

Parameters

record: dict: default=None
A logbook record

logbook:
Current stream logbook with the stats required

estimator:
GASearchCV Estimator that is being optimized

Returns

decision: bool, default=False
If `True`, the optimization process is stopped, else, if continues to the next generation.

`class sklearn_genetic.callbacks.`**`ProgressBar`**(***kwargs*)

Displays a tqdm progress bar with the training progress.

Parameters

kwargs: dict, default = {"file": sys.stdout}
A dict with valid arguments from tqdm.auto.tqdm

`on_end`(*logbook=None*, *estimator=None*)

Closes the progress bar

`on_start`(*estimator=None*)

Initializes the progress bar with the kwargs and total generations

`on_step`(*record=None*, *logbook=None*, *estimator=None*)

Increases the progress bar by one step

`class sklearn_genetic.callbacks.`**`ConsecutiveStopping`**(*generations*, *metric='fitness'*)

Stop the optimization if the current metric value is no greater that at least one metric from the last N generations

Parameters

generations: int, default=None
Number of current generations to compare against current generation

metric: {'fitness', 'fitness_std', 'fitness_max', 'fitness_min'}, default ='fitness'
Name of the metric inside 'record' logged in each iteration

`on_step`(*record=None*, *logbook=None*, *estimator=None*)

Take actions after fitting each generation.

Parameters

record: dict: default=None
A logbook record

logbook:
Current stream logbook with the stats required

estimator:
GASearchCV Estimator that is being optimized

Returns

decision: bool, default=False
If `True`, the optimization process is stopped, else, if continues to the next generation.

`class` sklearn_genetic.callbacks.**DeltaThreshold**(*threshold*, *generations=2*, *metric: str = 'fitness'*)

Stops the optimization if the absolute difference between the maximum and minimum value from the last N generations is less or equals to a threshold. The threshold gets evaluated after the number of generations specified is reached.

Parameters

threshold: float, default=None
Threshold to compare the differences between cross-validation scores.

generations: int, default=2
Number of generations to compare, includes the current generation.

metric: {'fitness', 'fitness_std', 'fitness_max', 'fitness_min'}, default ='fitness'
Name of the metric inside 'record' logged in each iteration.

on_step(*record=None*, *logbook=None*, *estimator=None*)

Take actions after fitting each generation.

Parameters

record: dict: default=None
A logbook record

logbook:
Current stream logbook with the stats required

estimator:
GASearchCV Estimator that is being optimized

Returns

decision: bool, default=False
If `True`, the optimization process is stopped, else, if continues to the next generation.

`class` sklearn_genetic.callbacks.**TimerStopping**(*total_seconds*)

Stops the optimization process if a limit training time has been elapsed. This time is checked after each generation fit

Parameters

total_seconds: int
Total time in seconds that the estimator is allowed to fit

on_start(*estimator=None*)

Take actions at the start of the training

Parameters

estimator:
GASearchCV Estimator that is being optimized

`on_step`(*record=None*, *logbook=None*, *estimator=None*)

Take actions after fitting each generation.

Parameters

record: dict: default=None
A logbook record

logbook:
Current stream logbook with the stats required

estimator:
GASearchCV Estimator that is being optimized

Returns

decision: bool, default=False
If `True`, the optimization process is stopped, else, if continues to the next generation.

`class` sklearn_genetic.callbacks.`ThresholdStopping`(*threshold*, *metric='fitness'*)

Stop the optimization if the metric from cross validation score is greater or equals than the define threshold

Parameters

threshold: float, default=None
Threshold to compare against the current cross validation average score and determine if the optimization process must stop

metric: {'fitness', 'fitness_std', 'fitness_max', 'fitness_min'}, default ='fitness'
Name of the metric inside 'record' logged in each iteration

`on_step`(*record=None*, *logbook=None*, *estimator=None*)

Take actions after fitting each generation.

Parameters

record: dict: default=None
A logbook record

logbook:
Current stream logbook with the stats required

estimator:
GASearchCV Estimator that is being optimized

Returns

decision: bool, default=False
If `True`, the optimization process is stopped, else, if continues to the next generation.

`class` sklearn_genetic.callbacks.`TensorBoard`(*log_dir='./logs'*, *run_id=None*)

Log all the fitness metrics to Tensorboard into log_dir/run_id folder

Parameters

log_dir: str, default="./logs"
Path to the main folder where the data will be log

run_id: str, default=None
Subfolder where the data will be log, if None it will create a folder with the current datetime with format time.strftime("%Y_%m_%d-%H_%M_%S")

on_step(*record=None*, *logbook=None*, *estimator=None*)

Take actions after fitting each generation.

Parameters

record: dict: default=None
A logbook record

logbook:
Current stream logbook with the stats required

estimator:
GASearchCV Estimator that is being optimized

Returns

decision: bool, default=False
If `True`, the optimization process is stopped, else, if continues to the next generation.

class sklearn_genetic.callbacks.**LogbookSaver**(*checkpoint_path*, ***dump_options*)

Saves the estimator.logbook parameter chapter object in a local file system.

Parameters

checkpoint_path: str
Location where checkpoint will be saved to

dump_options, str
Valid kwargs from joblib `dump`

on_step(*record=None*, *logbook=None*, *estimator=None*)

Take actions after fitting each generation.

Parameters

record: dict: default=None
A logbook record

logbook:
Current stream logbook with the stats required

estimator:
GASearchCV Estimator that is being optimized

Returns

decision: bool, default=False
If `True`, the optimization process is stopped, else, if continues to the next generation.

1.18 Schedules

base.BaseAdapter(initial_value, end_value, ...)	Base class for all the adapters
ConstantAdapter(initial_value, end_value, ...)	This adapter keep the current value equals to the initial_value it's mainly used to have an uniform interface when defining a parameter as constant vs as an adapter
ExponentialAdapter(initial_value, end_value, ...)	Adapts the initial value towards the end value using an exponential "decay" function
InverseAdapter(initial_value, end_value, ...)	Adapts the initial value towards the end value using a "decay" function of the form 1/x
PotentialAdapter(initial_value, end_value, ...)	Adapts the initial value towards the end value using a potential "decay" function

class sklearn_genetic.schedules.**BaseAdapter**(*initial_value*, *end_value*, *adaptive_rate*, ***kwargs*)

Base class for all the adapters

Parameters

initial_value
[float,] Initial value to be adapted

end_value
[float,] The final (asymptotic) value that the initial_value can take

adaptive_rate
[float,] Controls how fast the initial_value approaches the end_value

kwargs
[dict,] Possible extra parameters, None for now

Attributes

current_step
[int,] The current number of iterations that the adapter has run

current_value
[float,] The transformed initial_value after current_steps changes

abstract step()

Run one iteration of the transformation

class sklearn_genetic.schedules.**ConstantAdapter**(*initial_value*, *end_value*, *adaptive_rate*)

This adapter keep the current value equals to the initial_value it's mainly used to have an uniform interface when defining a parameter as constant vs as an adapter

Parameters

initial_value
[float,] Initial value to be adapted

end_value
[float,] The final (asymptotic) value that the initial_value can take

adaptive_rate
[float,] Controls how fast the initial_value approaches the end_value

Attributes

current_step
[int,] The current number of iterations that the adapter has run

current_value
[float,] Same as the initial_value

`step()`
Run one iteration of the transformation

`class sklearn_genetic.schedules.ExponentialAdapter(initial_value, end_value, adaptive_rate)`
Adapts the initial value towards the end value using an exponential "decay" function

Parameters

initial_value
[float,] Initial value to be adapted

end_value
[float,] The final (asymptotic) value that the initial_value can take

adaptive_rate
[float,] Controls how fast the initial_value approaches the end_value

Attributes

current_step
[int,] The current number of iterations that the adapter has run

current_value
[float,] The transformed initial_value after current_steps changes

`step()`
Run one iteration of the transformation

`class sklearn_genetic.schedules.InverseAdapter(initial_value, end_value, adaptive_rate)`
Adapts the initial value towards the end value using a "decay" function of the form 1/x

Parameters

initial_value
[float,] Initial value to be adapted

end_value
[float,] The final (asymptotic) value that the initial_value can take

adaptive_rate
[float,] Controls how fast the initial_value approaches the end_value

Attributes

current_step
[int,] The current number of iterations that the adapter has run

current_value
[float,] The transformed initial_value after current_steps changes

`step()`
Run one iteration of the transformation

`class sklearn_genetic.schedules.PotentialAdapter(initial_value, end_value, adaptive_rate)`
Adapts the initial value towards the end value using a potential "decay" function

Parameters

initial_value
[float,] Initial value to be adapted

end_value
[float,] The final (asymptotic) value that the initial_value can take

adaptive_rate
[float,] Controls how fast the initial_value approaches the end_value

Attributes

current_step
[int,] The current number of iterations that the adapter has run

current_value
[float,] The transformed initial_value after current_steps changes

`step()`
Run one iteration of the transformation

1.19 Plots

plot_fitness_evolution(estimator[, metric])	**Parameters**
plot_search_space(estimator[, height, s, ...])	**Parameters**

sklearn_genetic.plots.**plot_fitness_evolution**(*estimator*, *metric='fitness'*)

Parameters

estimator: estimator object
A fitted estimator from *GASearchCV*

metric: {"fitness", "fitness_std", "fitness_max", "fitness_min"}, default="fitness"
Logged metric into the estimator history to plot

Returns

Lines plot with the fitness value in each generation

sklearn_genetic.plots.**plot_search_space**(*estimator*, *height=2*, *s=25*, *features: list = None*)

Parameters

estimator: estimator object
A fitted estimator from *GASearchCV*

height: float, default=2
Height of each facet

s: float, default=5
Size of the markers in scatter plot

features: list, default=None
Subset of features to plot, if `None` it plots all the features by default

Returns

Pair plot of the used hyperparameters during the search

1.20 MLflow

class sklearn_genetic.mlflow_log.**MLflowConfig**(*tracking_uri*, *experiment*, *run_name*, *save_models=False*, *registry_uri=None*, *tags=None*)

Logs each fit of hyperparameters in a running instance of mlflow: https://mlflow.org/

Parameters

tracking_uri: str
Address of local or remote tracking server.

experiment: str
Case sensitive name of an experiment to be activated.

run_name: str
Name of new run (stored as a mlflow.runName tag).

save_models: bool, default=False
If `True`, it will log the estimator into mlflow artifacts

registry_uri: str, default=None
Address of local or remote model registry server.

tags: dict, default=None
Dictionary of tag_name: String -> value.

create_run(*parameters*, *score*, *estimator*)

Parameters

parameters: dict
A dictionary with the keys as the hyperparameter name and the value as the current value setting

score:
The cross-validation score achieved by the current parameters

estimator: estimator object
The current sklearn estimator that is being fitted

1.21 Space

class sklearn_genetic.space.**Categorical**(*choices: list = None*, *priors: list = None*, *distribution: str = 'choice'*, *random_state=None*)

class for hyperparameters search space of categorical values

Parameters

choices: list, default=None
List with all the possible values of the hyperparameter.

priors: int, default=None
List with the probability of sampling each element of the "choices", if not set gives equals probability.

distribution: str, default='choice'
Distribution to sample initial population and mutation values, currently only supports "choice".

random_state
[int or None, RandomState instance, default=None] Pseudo random number generator state used for random dimension sampling.

`sample()`
Sample a random value from the assigned distribution

`class sklearn_genetic.space.Continuous(lower: float = None, upper: float = None, distribution: str = 'uniform', random_state=None)`
class for hyperparameters search space of real values

Parameters

lower
[int, default=None] Lower bound of the possible values of the hyperparameter.

upper
[int, default=None] Upper bound of the possible values of the hyperparameter.

distribution
[{'uniform', 'log-uniform'}, default='uniform'] Distribution to sample initial population and mutation values.

random_state
[int or None, RandomState instance, default=None] Pseudo random number generator state used for random dimension sampling.

`sample()`
Sample a random value from the assigned distribution

`class sklearn_genetic.space.Integer(lower: int = None, upper: int = None, distribution: str = 'uniform', random_state=None)`
class for hyperparameters search space of integer values

Parameters

lower
[int, default=None] Lower bound of the possible values of the hyperparameter.

upper
[int, default=None] Upper bound of the possible values of the hyperparameter.

distribution
[str, default="uniform"] Distribution to sample initial population and mutation values, currently only supports 'uniform'.

random_state
[int or None, RandomState instance, default=None] Pseudo random number generator state used for random dimension sampling.

`sample()`
Sample a random value from the assigned distribution

`class sklearn_genetic.space.Space(param_grid: dict = None)`
Search space for all the models hyperparameters

Parameters

param_grid: dict, default=None
Grid with the parameters to tune, expects keys a valid name of hyperparameter based on the estimator selected and as values one of *Integer* , *Categorical Continuous* classes

property dimensions

Returns

The number of hyperparameters defined in the param_grid

property parameters

Returns

A list with all the names of the hyperparametes in the param_Grid

1.22 Algorithms

eaMuPlusLambda(population, toolbox, mu, ...)	The base implementation is directly taken from: https://github.com/DEAP/deap/blob/master/deap/algorithms.py
eaMuCommaLambda(population, toolbox, mu, ...)	The base implementation is directly taken from: https://github.com/DEAP/deap/blob/master/deap/algorithms.py
eaSimple(population, toolbox, cxpb, mutpb, ngen)	The base implementation is directly taken from: https://github.com/DEAP/deap/blob/master/deap/algorithms.py

sklearn_genetic.algorithms.**eaMuCommaLambda**(*population*, *toolbox*, *mu*, *lambda_*, *cxpb*, *mutpb*, *ngen*, *stats=None*, *halloffame=None*, *callbacks=None*, *verbose=True*, *estimator=None*)

The base implementation is directly taken from: https://github.com/DEAP/deap/blob/master/deap/algorithms.py

This is the $(\mu\ ,\ \lambda)$ evolutionary algorithm.

population: A list of individuals.
Population resulting of the iteration process.

toolbox: A Toolbox
Contains the evolution operators.

mu: int, default=None,
The number of individuals to select for the next generation.

lambda_: int, default=None
The number of children to produce at each generation.

cxpb: Scheduler, default=None
The probability that an offspring is produced by crossover.

mutpb: Scheduler, default=None
An adaptive scheduler representing the probability that an offspring is produced by mutation.

ngen: int, default=None
The number of generation.

stats: A Statistics
Object that is updated inplace, optional.

halloffame: A HallOfFame
Object that will contain the best individuals, optional.

callbacks: list or Callable
One or a list of the `callbacks` methods available in the package.

verbose: bool, default=True
Whether or not to log the statistics.

estimator: *GASearchCV*, default = None
Estimator that is being optimized

Returns

pop: list
The final population.

log: Logbook
Statistics of the evolution.

n_gen: int
Number of generations used.

sklearn_genetic.algorithms.**eaMuPlusLambda**(*population*, *toolbox*, *mu*, *lambda_*, *cxpb*, *mutpb*, *ngen*, *stats=None*, *halloffame=None*, *callbacks=None*, *verbose=True*, *estimator=None*)

The base implementation is directly taken from: https://github.com/DEAP/deap/blob/master/deap/algorithms.py

This is the $(\mu + \lambda)$ evolutionary algorithm.

population: A list of individuals.
Population resulting of the iteration process.

toolbox: A `Toolbox`
Contains the evolution operators.

mu: int, default=None
The number of individuals to select for the next generation.

lambda_: int, default=None
The number of children to produce at each generation.

cxpb: Scheduler, default=None
The probability that an offspring is produced by crossover.

mutpb: Scheduler, default=None
An adaptive scheduler representing the probability that an offspring is produced by mutation.

ngen: int, default=None
The number of generation.

stats: A `Statistics`
Object that is updated inplace, optional.

halloffame: A `HallOfFame`
Object that will contain the best individuals, optional.

callbacks: list or Callable
One or a list of the `callbacks` methods available in the package.

verbose: bool, default=True
Whether or not to log the statistics.

estimator: *GASearchCV*, default = None
Estimator that is being optimized

Returns

pop: list
The final population.

log: Logbook
Statistics of the evolution.

n_gen: int
Number of generations used.

sklearn_genetic.algorithms.**eaSimple**(*population*, *toolbox*, *cxpb*, *mutpb*, *ngen*, *stats=None*, *halloffame=None*, *callbacks=None*, *verbose=True*, *estimator=None*)

The base implementation is directly taken from: https://github.com/DEAP/deap/blob/master/deap/algorithms.py

This algorithm reproduce the simplest evolutionary algorithm as presented in chapter 7 of Back2000.

population: A list of individuals.
Population resulting of the iteration process.

toolbox: A `Toolbox`
Contains the evolution operators.

cxpb: Scheduler, default=None
An adaptive scheduler representing the probability of mating two individuals.

mutpb: Scheduler, default=None
An adaptive scheduler representing the probability that an offspring is produced by mutation.

ngen: int, default=None
The number of generation.

stats: A `Statistics`
Object that is updated inplace, optional.

halloffame: A `HallOfFame`
Object that will contain the best individuals, optional.

callbacks: list or callable
One or a list of the `callbacks` methods available in the package.

verbose: bool, default=True
Whether or not to log the statistics.

estimator: *GASearchCV*, default = None
Estimator that is being optimized

Returns

pop: list
The final population.

log: Logbook
Statistics of the evolution.

n_gen: int
Number of generations used.

These external references show some examples, tutorials or use cases of Sklearn-genetic-opt.

1.23 Articles

- Evolutionary Feature Selection for Machine Learning
- Hyperparameters Tuning: From Grid Search to Optimization.
- Tune your Scikit-learn model using evolutionary algorithms

1.24 Contributing

You can add your external reference or contribute to this package by checking the contributing guides.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

S

INDEX

P

S

T