How to Use Sklearn-genetic-opt

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.

Fine-tuning Example

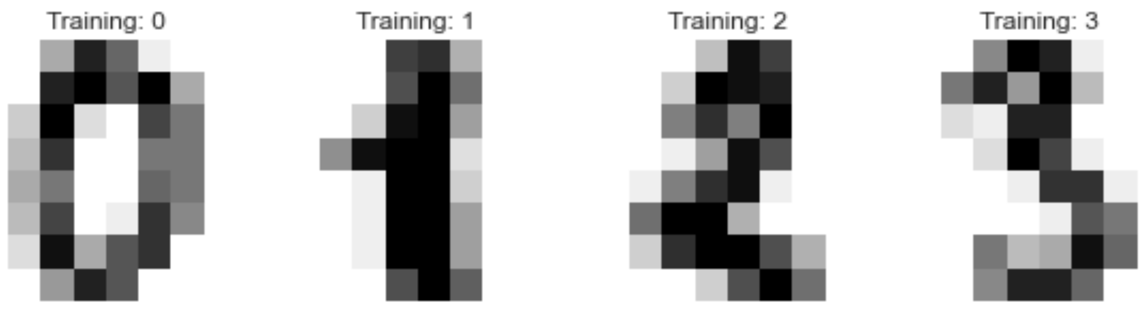
First let's import some dataset and other scikit-learn standard modules, we'll use the digits dataset. This is a classification problem, we'll fine-tune a Random Forest Classifier for this task.

```
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:

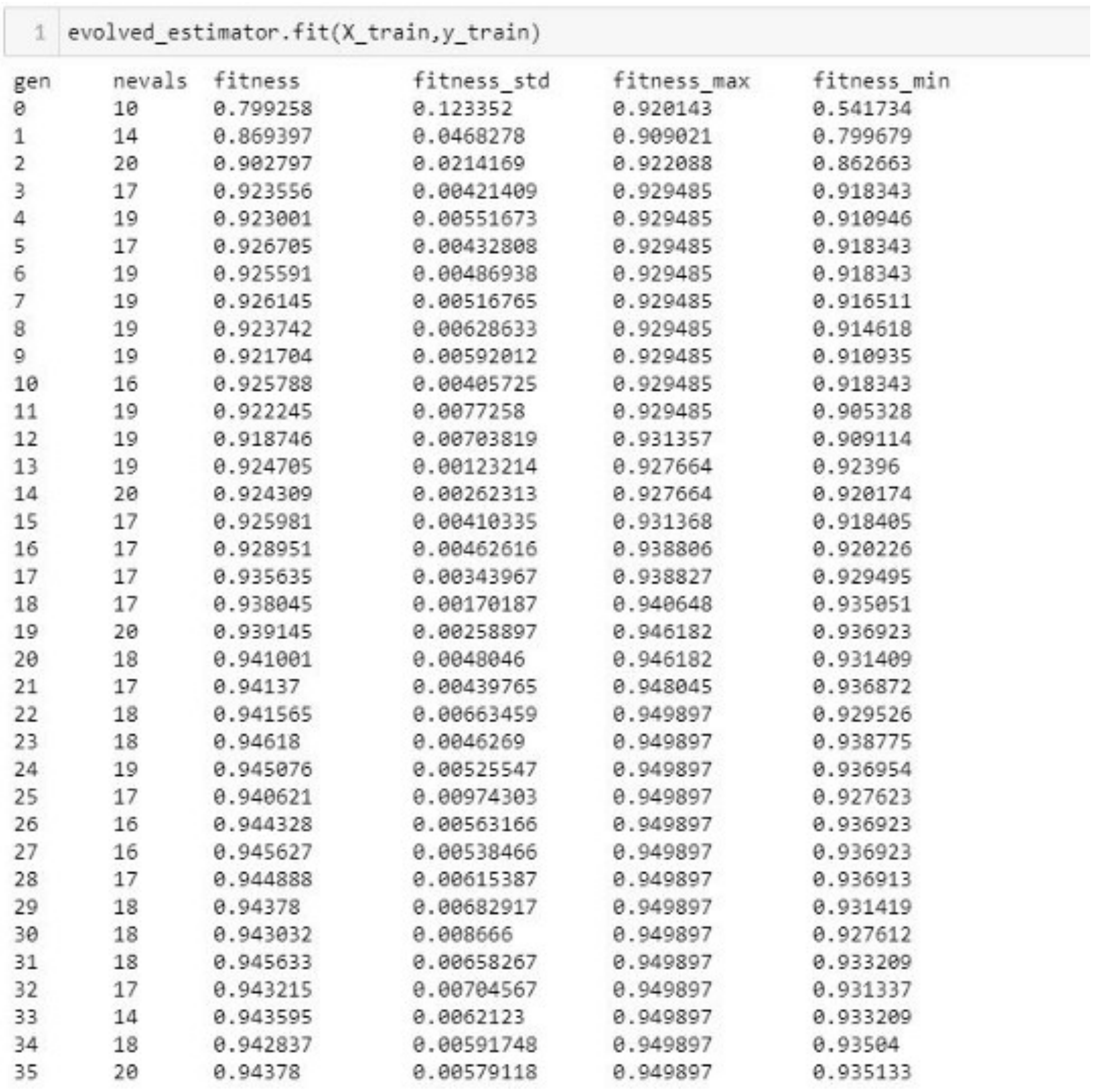
```
# The base classifier to tune
clf = RandomForestClassifier()

# 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:



```python
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   
```

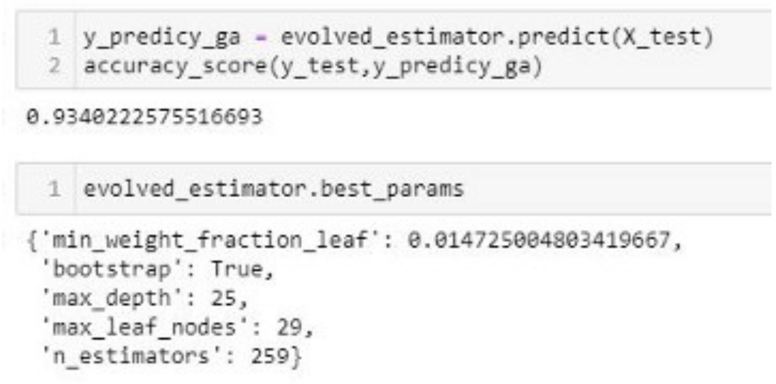
This log, shows us the metrics obtained in each iteration (generation), this is what each entry means:

- **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:

```python
# 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)
```

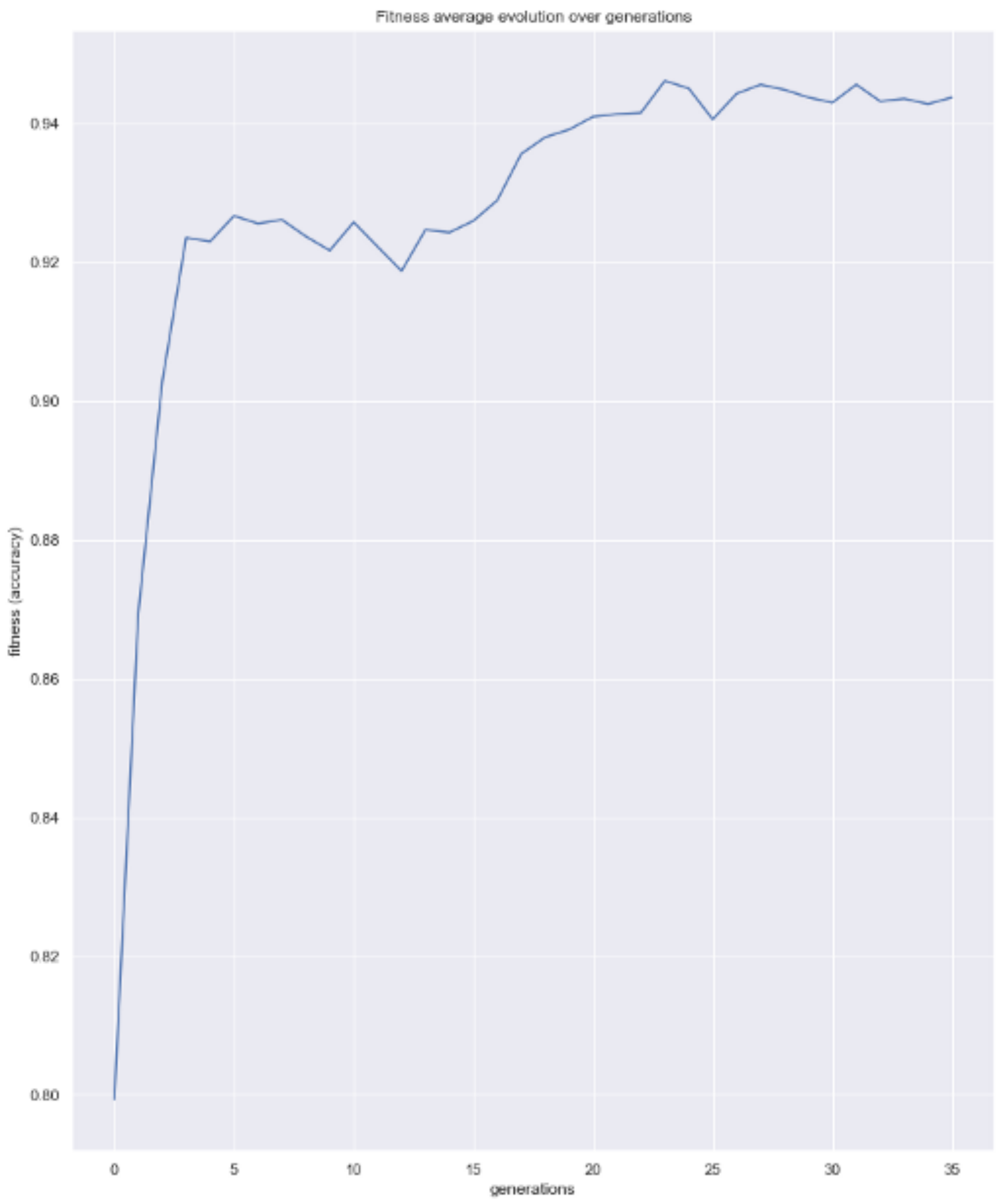
```
0.9340222575516693
```

```
evolved_estimator.best_params
```

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

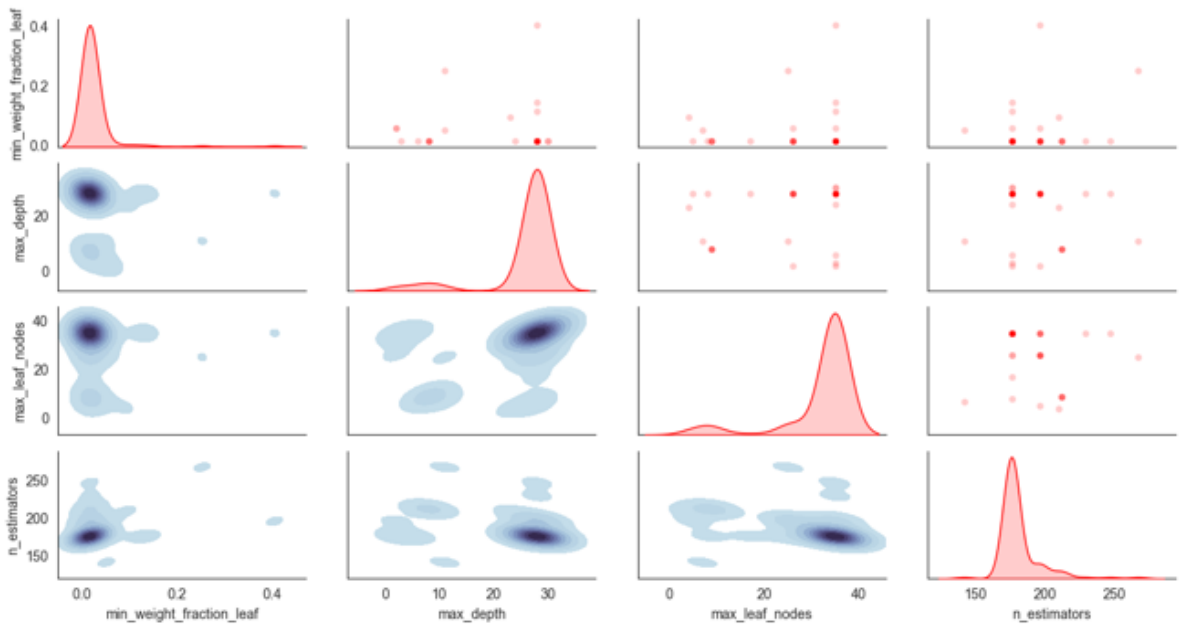
Now, let's use a couple more functions available in the package. The first one will help us to see the evolution of our metric over the generations

```python
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:

```python
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.

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:

```python
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.

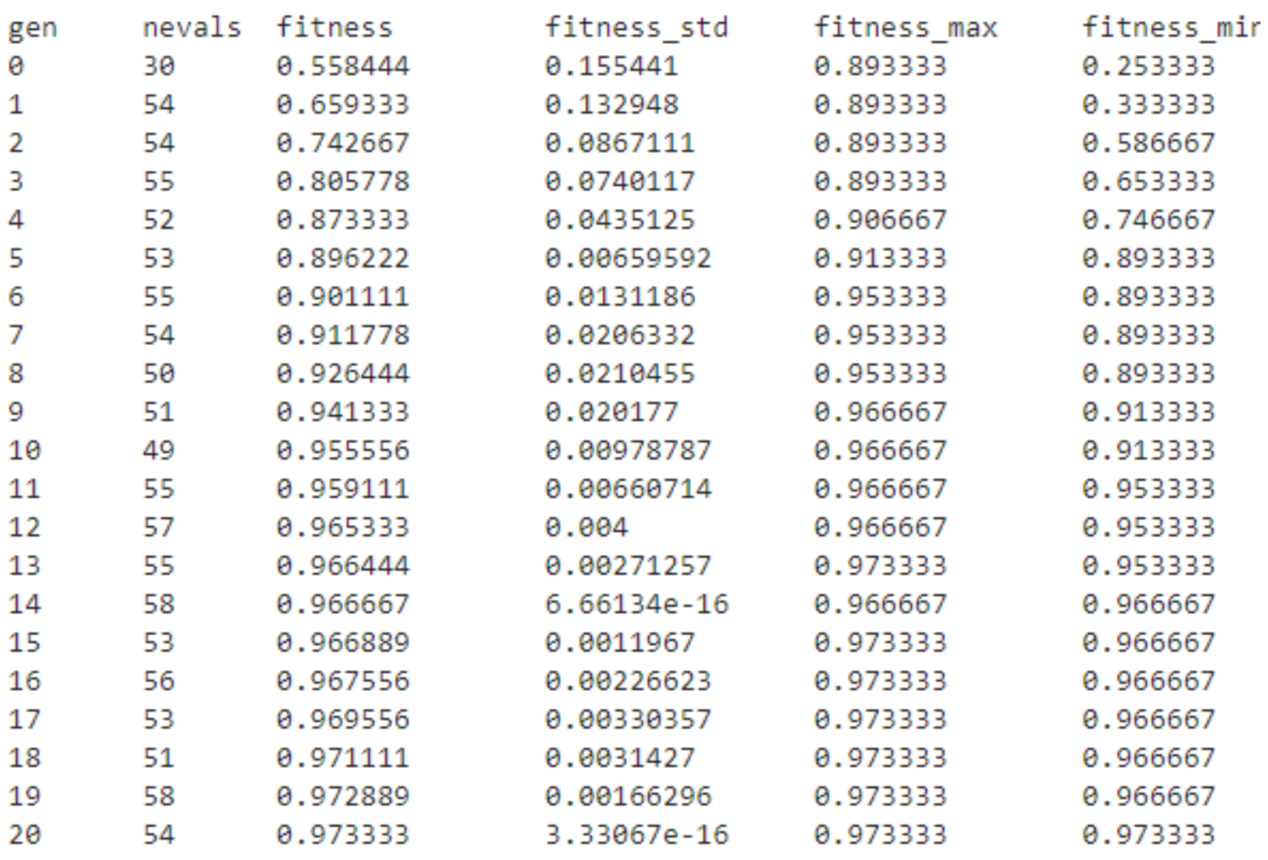
Let's create the feature selection object, the estimator we're going to use is a SVM:

```
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:

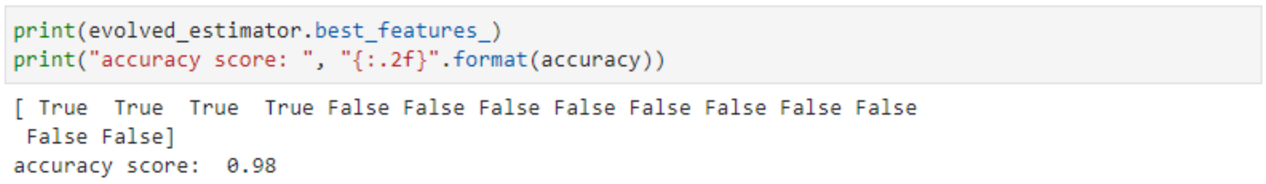
```
# Train and select the features
evolved_estimator.fit(X_train, y_train)
```

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:

```python
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)
```

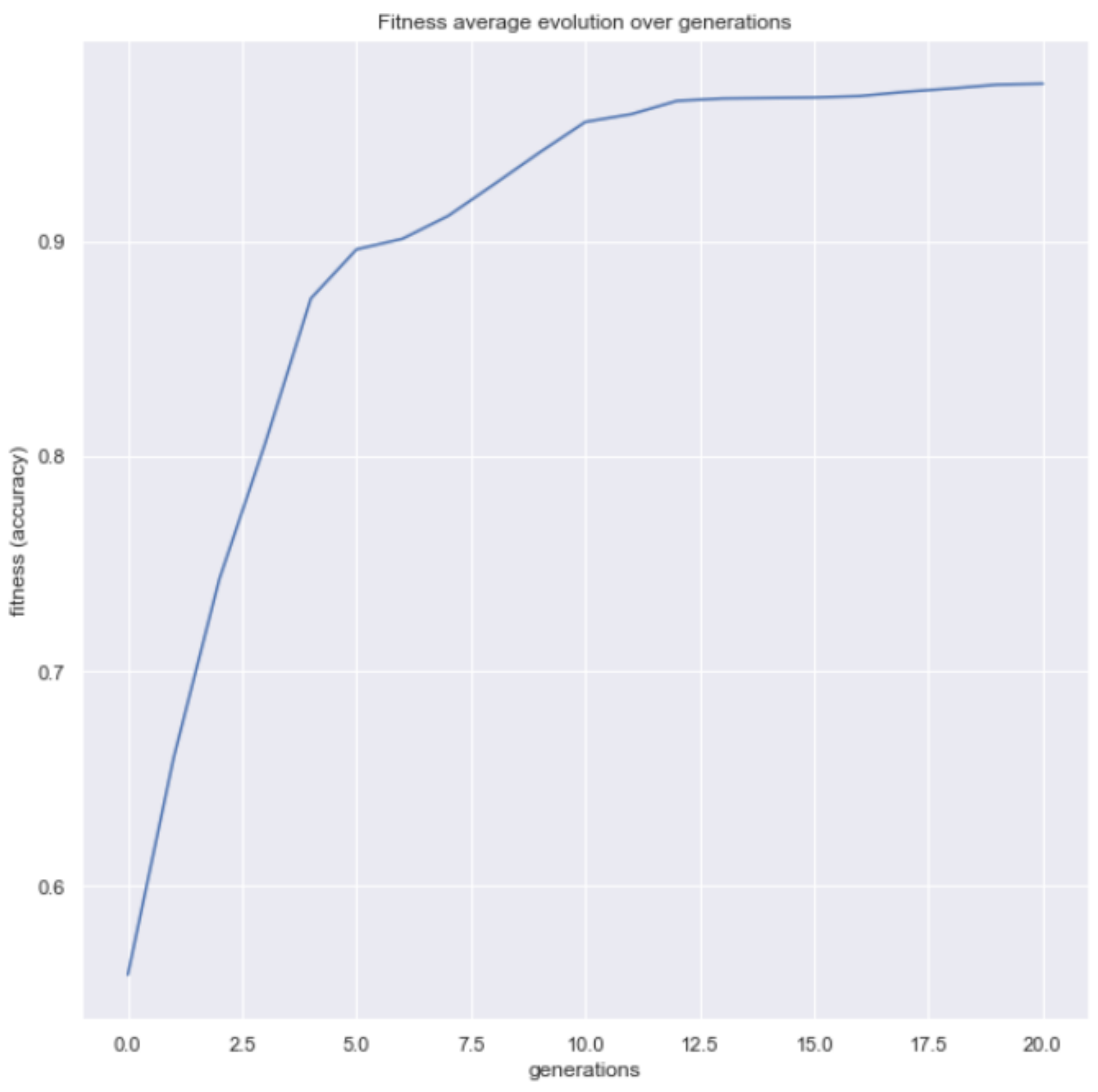

```
print(evolved_estimator.best_features_)
print("accuracy score: ", "{:.2f}".format(accuracy))
```
```
[ True  True  True  True False False False False False False False False
 False False]
accuracy score:  0.98
```

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:

```python
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.