DATA ANALYSIS

A complete Data Analysis workflow in Python and scikit-learn

A ready-to-run code including preprocessing, parameters tuning and model running and evaluation.

Angelica Lo Duca May 3 · 6 min read ★

Image by Buffik from Pixabay

In this short tutorial I illustrate a complete data analysis process which exploits the `scikit-learn` Python library. The process includes

- model selection with parameters tuning
- model evaluation

The code of this tutorial can be downloaded from my Github Repository.

Load Dataset

Firstly, I load the dataset through the Python `pandas` library. I exploit the `heart.csv` dataset, provided by the Kaggle repository.

```
import pandas as pd

df = pd.read_csv('source/heart.csv')
df.head()
```

	age	sex	cp	trtbps	chol	fbs	restecg	thalachh	exng	oldpeak	slp	caa	thall	output
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1

Image by Author

I calculate the number of records and the number of columns in the dataset:

```
df.shape
```

which gives the following output:

```
(303, 14)
```

Features selection

```
features = []
for column in df.columns:
    if column != 'output':
        features.append(column)
X = df[features]
Y = df['output']
```

In order to select the minimum set of input features, I calculate the Pearson correlation coefficient among features, through `corr()` function, provided by a `pandas dataframe`.

	age	sex	cp	trtbps	chol	fbs	restecg	thalachh	exng	oldpeak	slp	caa	thall
age	1.000000	-0.098447	-0.068653	0.279351	0.213678	0.121308	-0.116211	-0.398522	0.096801	0.210013	-0.168814	0.276326	0.068001
sex	-0.098447	1.000000	-0.049353	-0.056769	-0.197912	0.045032	-0.058196	-0.044020	0.141664	0.096093	-0.030711	0.118261	0.210041
cp	-0.068653	-0.049353	1.000000	0.047608	-0.076904	0.094444	0.044421	0.295762	-0.394280	-0.149230	0.119717	-0.181053	-0.161736
trtbps	0.279351	-0.056769	0.047608	1.000000	0.123174	0.177531	-0.114103	-0.046698	0.067616	0.193216	-0.121475	0.101389	0.062210
chol	0.213678	-0.197912	-0.076904	0.123174	1.000000	0.013294	-0.151040	-0.009940	0.067023	0.053952	-0.004038	0.070511	0.098803
fbs	0.121308	0.045032	0.094444	0.177531	0.013294	1.000000	-0.084189	-0.008567	0.025665	0.005747	-0.059894	0.137979	-0.032019
restecg	-0.116211	-0.058196	0.044421	-0.114103	-0.151040	-0.084189	1.000000	0.044123	-0.070733	-0.058770	0.093045	-0.072042	-0.011981
thalachh	-0.398522	-0.044020	0.295762	-0.046698	-0.009940	-0.008567	0.044123	1.000000	-0.378812	-0.344187	0.386784	-0.213177	-0.096439
exng	0.096801	0.141664	-0.394280	0.067616	0.067023	0.025665	-0.070733	-0.378812	1.000000	0.288223	-0.257748	0.115739	0.206754
oldpeak	0.210013	0.096093	-0.149230	0.193216	0.053952	0.005747	-0.058770	-0.344187	0.288223	1.000000	-0.577537	0.222682	0.210244
slp	-0.168814	-0.030711	0.119717	-0.121475	-0.004038	-0.059894	0.093045	0.386784	-0.257748	-0.577537	1.000000	-0.080155	-0.104764
caa	0.276326	0.118261	-0.181053	0.101389	0.070511	0.137979	-0.072042	-0.213177	0.115739	0.222682	-0.080155	1.000000	0.151832
thall	0.068001	0.210041	-0.161736	0.062210	0.098803	-0.032019	-0.011981	-0.096439	0.206754	0.210244	-0.104764	0.151832	1.000000

Image by Author

I note that all the features have a low correlation, thus I can keep all of them as input features.

Data Normalization

Data Normalization scales all the features in the same interval. I exploit the `MinMaxScaler()` provided by the `scikit-learn` library. I dealt with Data Normalization in `scikit-learn` in my previous article, while I this article I described the general process of Data Normalization without `scikit-learn`.

```
X.describe()
```

Get started Open in app

min	29.000000	0.000000	0.000000	94.000000	126.000000	0.000000	0.000000	71.000000	0.000000	0.000000	0.000000	0.000000	0.0
25%	47.500000	0.000000	0.000000	120.000000	211.000000	0.000000	0.000000	133.500000	0.000000	0.000000	1.000000	0.000000	2.0
50%	55.000000	1.000000	1.000000	130.000000	240.000000	0.000000	1.000000	153.000000	0.000000	0.800000	1.000000	0.000000	2.0
75%	61.000000	1.000000	2.000000	140.000000	274.500000	0.000000	1.000000	166.000000	1.000000	1.600000	2.000000	1.000000	3.0
max	77.000000	1.000000	3.000000	200.000000	564.000000	1.000000	2.000000	202.000000	1.000000	6.200000	2.000000	4.000000	3.0

Image by Author

Looking at the minimum and maximum value for each feature, I note that there are many features out the range [0,1], thus I need to scale them.

For each input feature I calculate the `MinMaxScaler()` and I store the result in the same `X` column. The `MinMaxScaler()` must be fitted firstly through the `fit()` function and then can be applied for a transformation through the `transform()` function. Note that I must reshape every feature in the format (-1,1) in order to be passed as input parameter of the scaler. For example, `Reshape(-1,1)` transforms the array `[0,1,2,3,5]` into `[[0],[1],[2],[3],[5]]`.

```
from sklearn.preprocessing import MinMaxScaler

for column in X.columns:
    feature = np.array(X[column]).reshape(-1,1)
    scaler = MinMaxScaler()
    scaler.fit(feature)
    feature_scaled = scaler.transform(feature)
    X[column] = feature_scaled.reshape(1,-1)[0]
```

Split the dataset in Training and Test

Now I split the dataset into two parts: training and testset. The test set size is 20% of the whole dataset. I exploit the `scikit-learn` function `train_test_split()`. I will use the training set to train the model and the testset to test the performance of the model.

```
import numpy as np
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split( X, Y,
test_size=0.20, random_state=42)
```

Balancing

number of records in each output class.

```
y_train.value_counts()
```

which gives the following output:

```
1    133
0    109
```

The output classes are not balanced, thus I can balance it. I can exploit the `imblearn` library, to perform balancing. I try both oversampling the minority class and undersampling the majority class. More details related to the Imbalanced Learn library can be found here. Firstly, I perform over sampling through the `RandomOverSampler()`. I create the model and then I fit with the training set. The `fit_resample()` function returns the balanced training set.

```
from imblearn.over_sampling import RandomOverSampler
over_sampler = RandomOverSampler(random_state=42)
X_bal_over, y_bal_over = over_sampler.fit_resample(X_train, y_train)
```

I calculate the number of records in each class through the `value_counts()` function and I note that now the dataset is balanced.

```
y_bal_over.value_counts()
```

which gives the following output:

```
1    133
0    133
```

Secondly, I perform under sampling through the `RandomUnderSampler()` model.

```
under_sampler = RandomUnderSampler(random_state=42)
X_bal_under, y_bal_under = under_sampler.fit_resample(X_train,
y_train)
```

Model Selection and Training

Now, I'm ready to train the model. I choose a `KNeighborsClassifier` and firstly I train it with imbalanced data. I exploit the `fit()` function to train the model and then the `predict_proba()` function to predict the values of the test set.

```
from sklearn.neighbors import KNeighborsClassifier

model = KNeighborsClassifier(n_neighbors=3)
model.fit(X_train, y_train)
y_score = model.predict_proba(X_test)
```

I calculate the performance of the model. In particular, I calculate the `roc_curve()` and the `precision_recall()` and then I plot them. I exploit the `scikitplot` library to plot curves.

From the plot I note that there is a roc curve for each class. With respect to the precision recall curve, the class 1 works better than class 0, probably because it is represented by a greater number of samples.

```
import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve
from scikitplot.metrics import plot_roc,auc
from scikitplot.metrics import plot_precision_recall

fpr0, tpr0, thresholds = roc_curve(y_test, y_score[:, 1])

# Plot metrics
plot_roc(y_test, y_score)
plt.show()

plot_precision_recall(y_test, y_score)
plt.show()
```

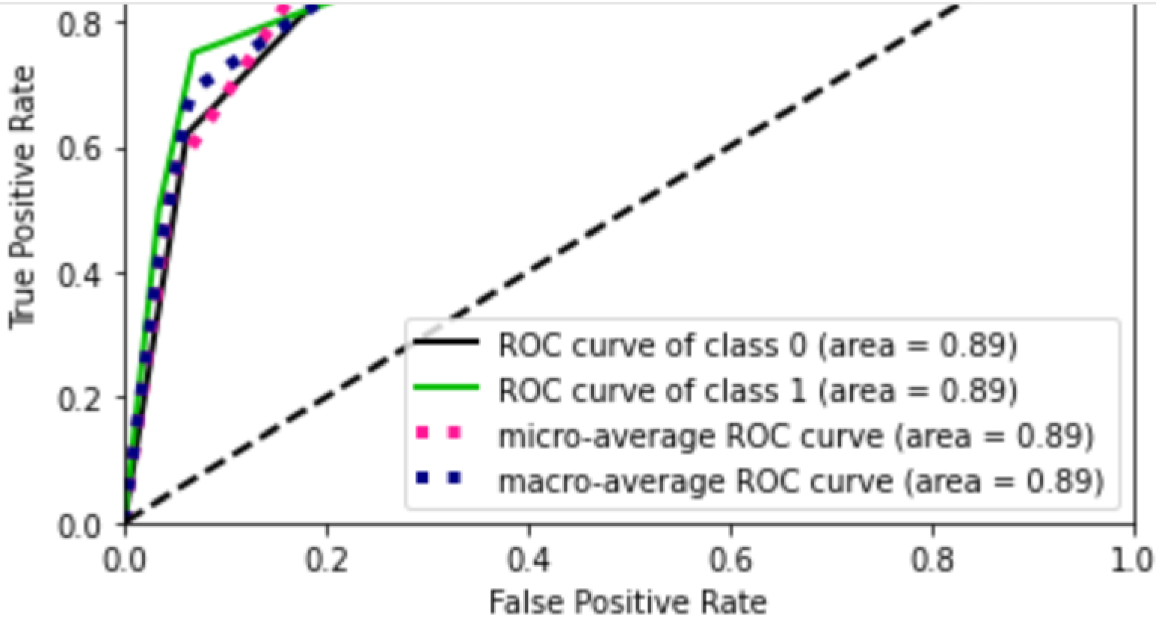


Image by Author

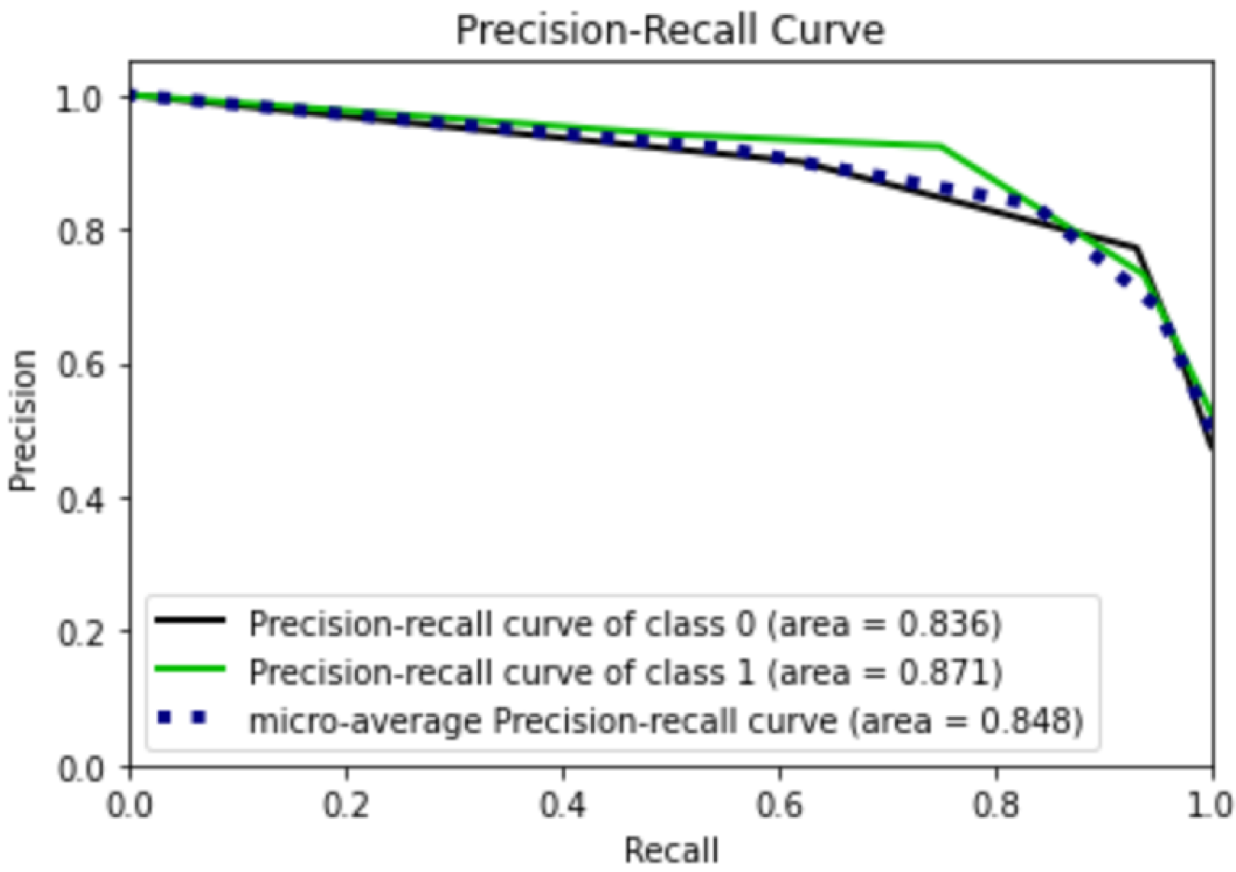


Image by Author

```
model = KNeighborsClassifier(n_neighbors=3)
model.fit(X_bal_over, y_bal_over)
y_score = model.predict_proba(X_test)
fpr0, tpr0, thresholds = roc_curve(y_test, y_score[:, 1])

# Plot metrics
plot_roc(y_test, y_score)
plt.show()

plot_precision_recall(y_test, y_score)
plt.show()
```

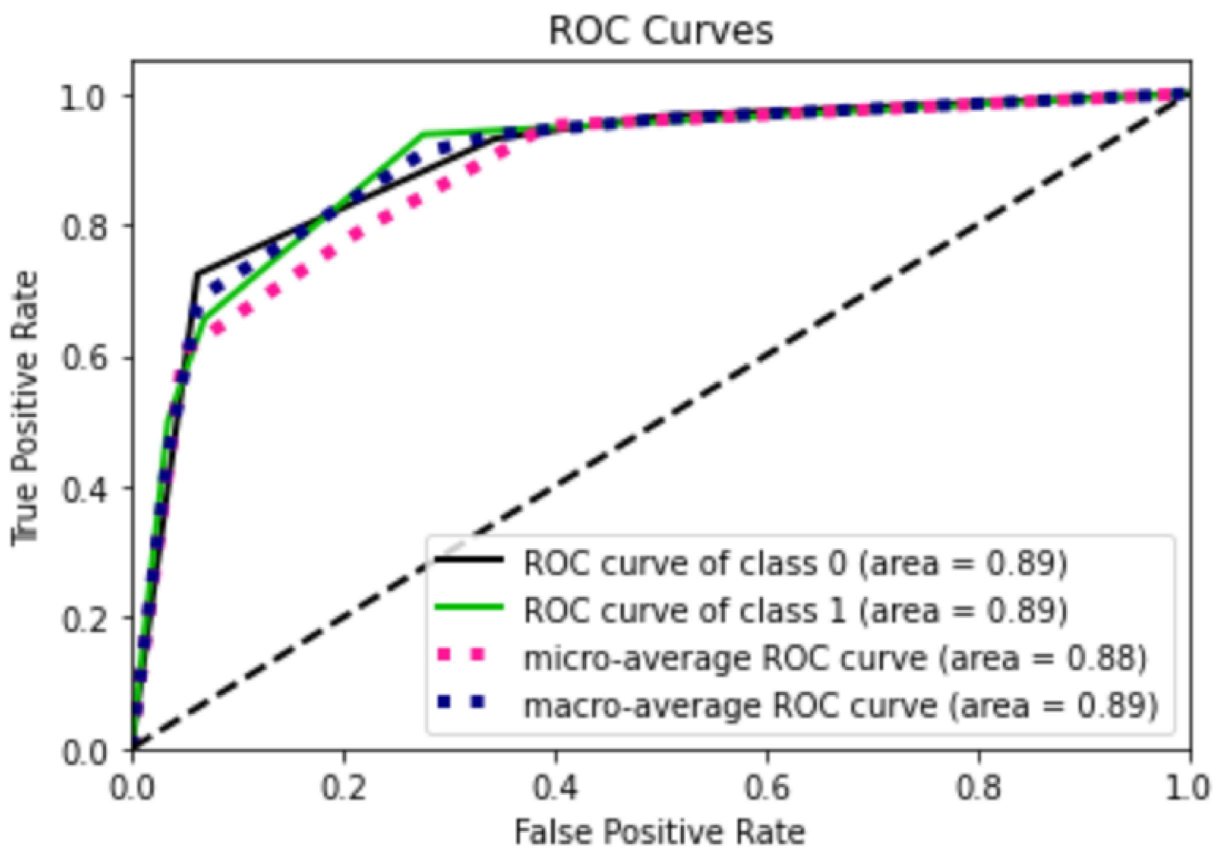
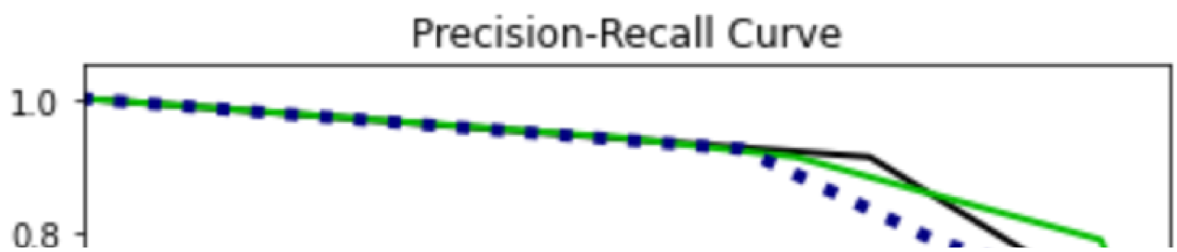


Image by Author



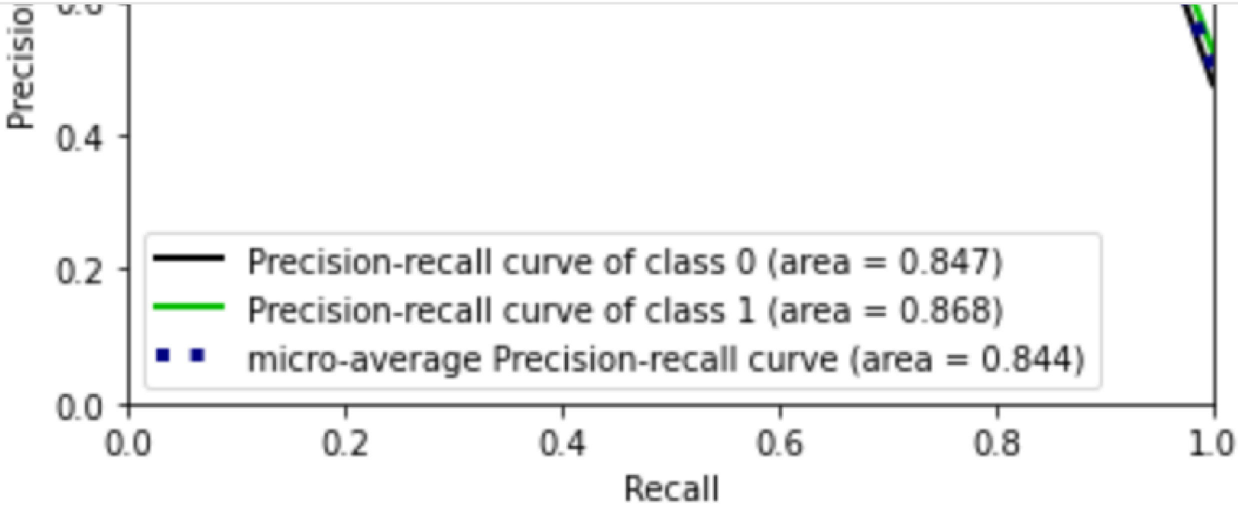


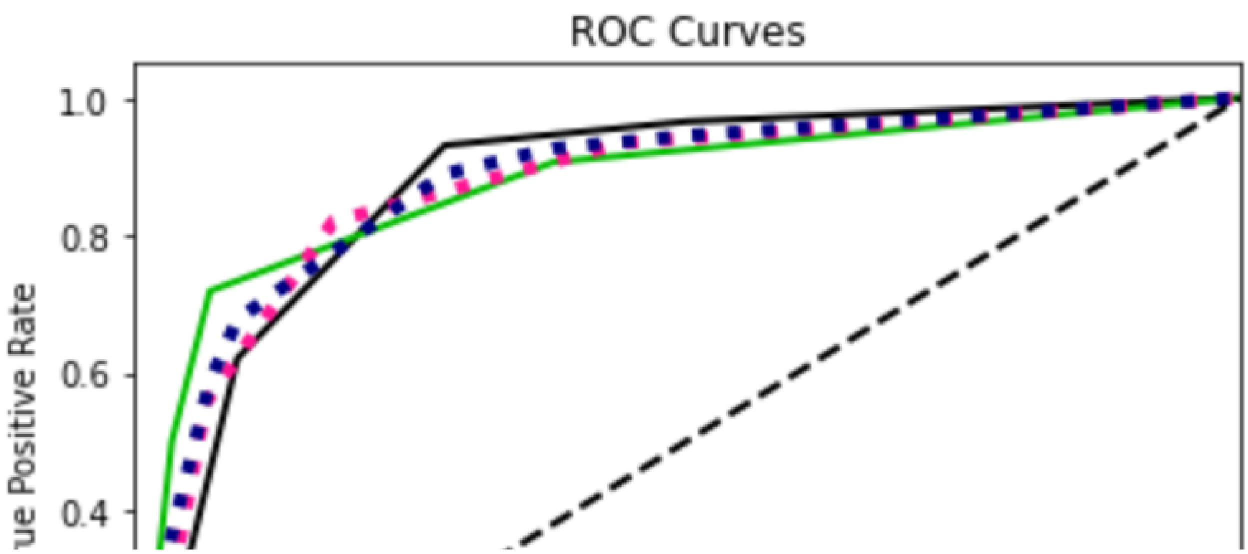
Image by Author

Finally, I train the model through under sampled data and I note a general deterioration of the performance.

```
model = KNeighborsClassifier(n_neighbors=3)
model.fit(X_bal_under, y_bal_under)
y_score = model.predict_proba(X_test)
fpr0, tpr0, thresholds = roc_curve(y_test, y_score[:, 1])

# Plot metrics
plot_roc(y_test, y_score)
plt.show()

plot_precision_recall(y_test, y_score)
plt.show()
```



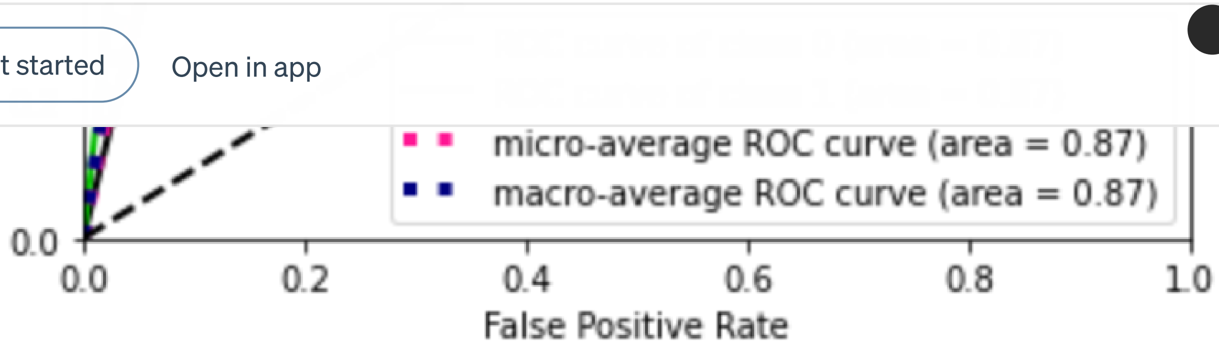


Image by Author

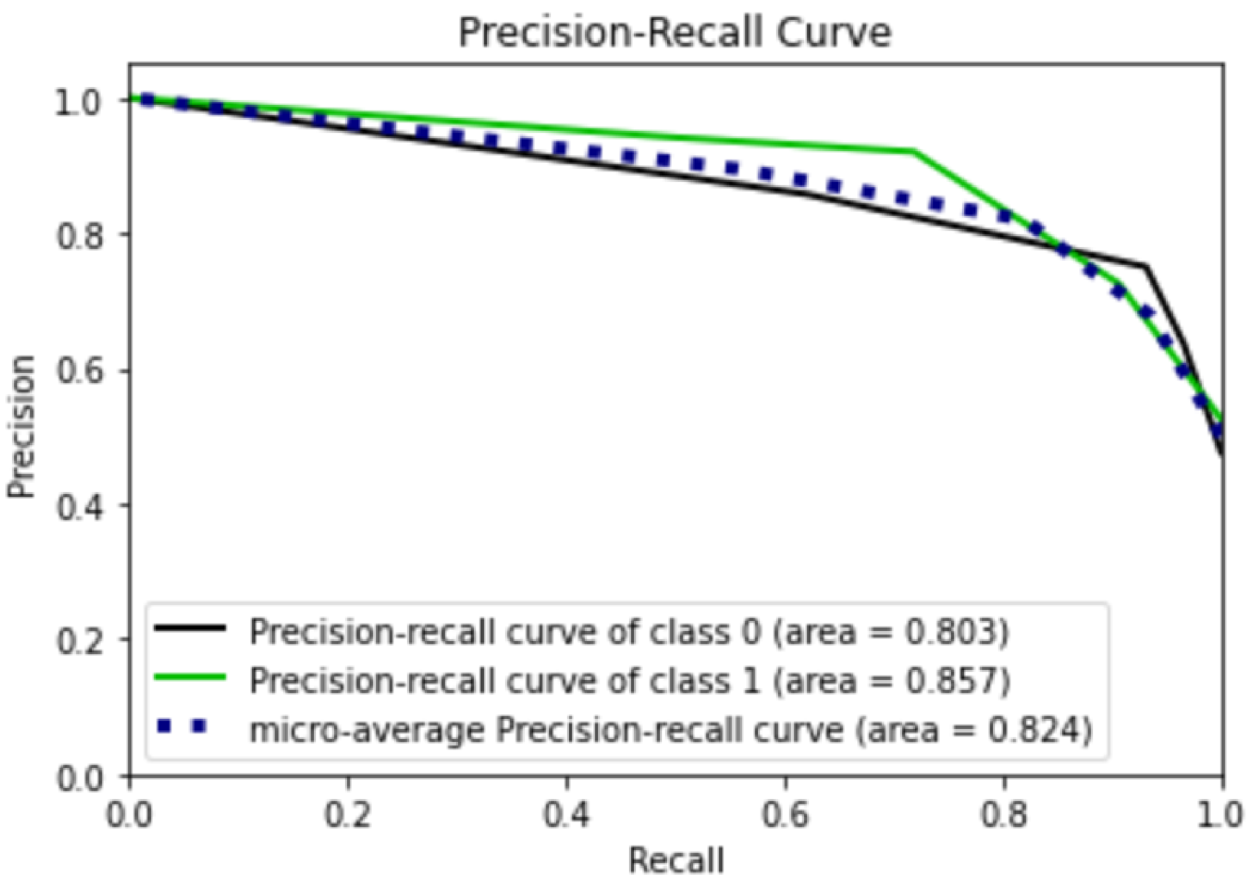


Image by Author

Parameters Tuning

In the last part of this tutorial, I try to improve the performance of the model by searching for best parameters for my model. I exploit the `GridSearchCV` mechanism provided by the `scikit-learn` library. I select a range of values for each parameter to be tested and I put them in the `param_grid` variable. I create a `GridSearchCV()` object, I fit with the training set and then I retrieve the best estimator, contained in the `best_estimator_` variable.

Get started Open in app

```
model = KNeighborsClassifier()

param_grid = {
   'n_neighbors': np.arange(2,8),
   'algorithm' : ['auto', 'ball_tree', 'kd_tree', 'brute'],
    'metric' : ['euclidean','manhattan','chebyshev','minkowski']
}

grid = GridSearchCV(model, param_grid = param_grid)
grid.fit(X_train, y_train)

best_estimator = grid.best_estimator_
```

I exploit the best estimator as model for my predictions and I calculate the performance of the algorithm.

```
best_estimator.fit(X_train, y_train)
y_score = best_estimator.predict_proba(X_test)
fpr0, tpr0, thresholds = roc_curve(y_test, y_score[:, 1])

# Plot metrics
plot_roc(y_test, y_score)
plt.show()

plot_precision_recall(y_test, y_score)
plt.show()
```

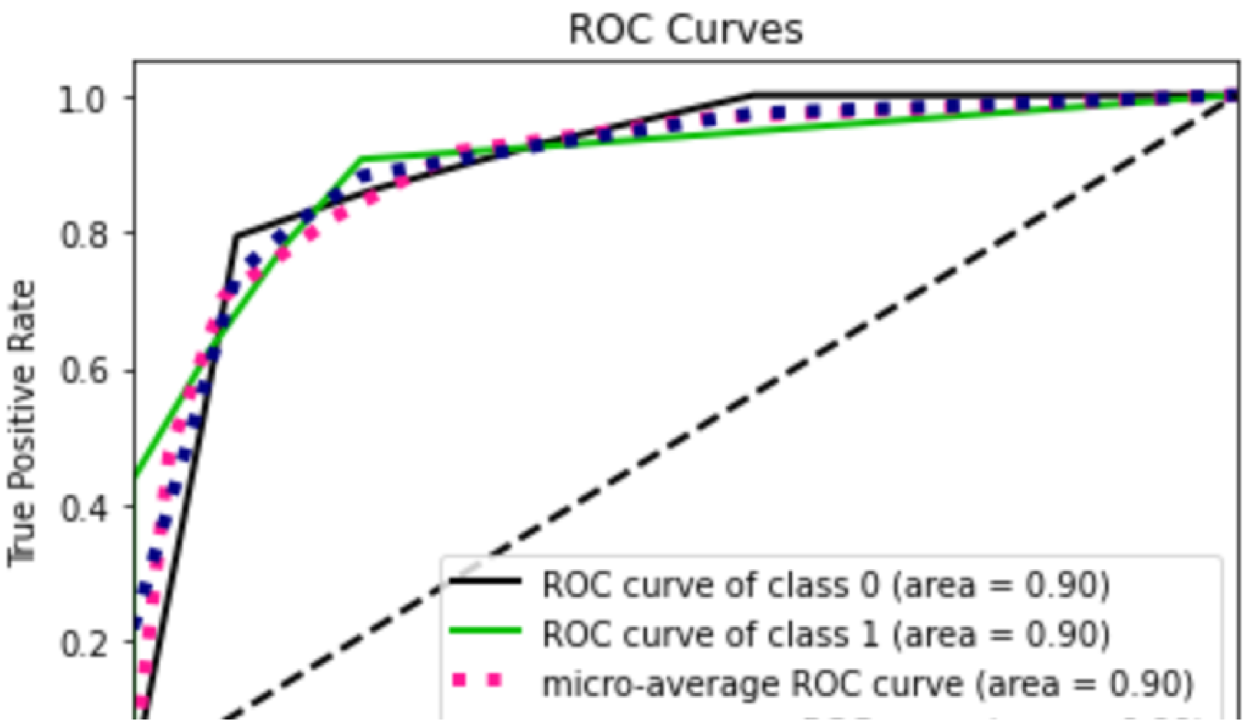


Image by Author

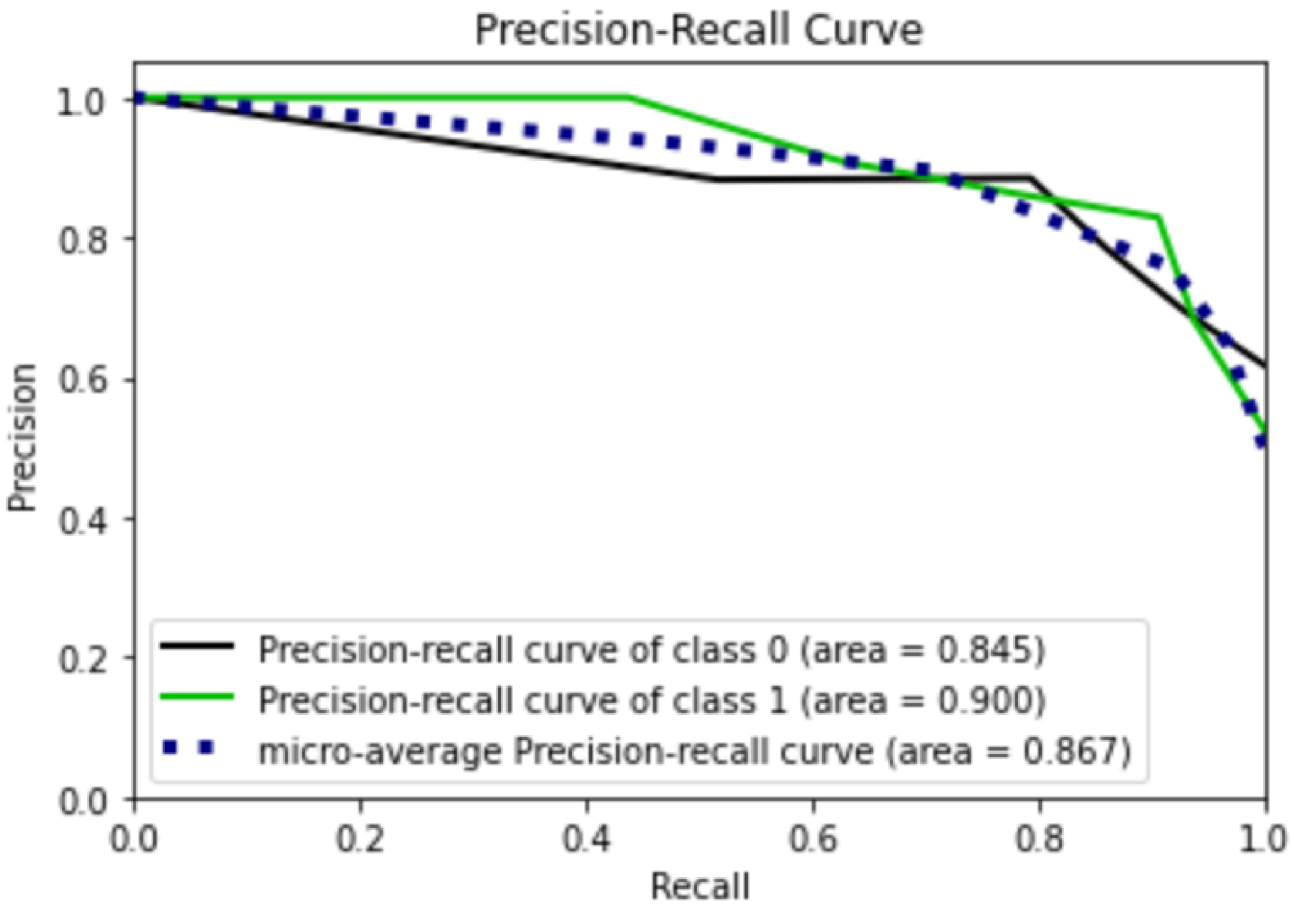
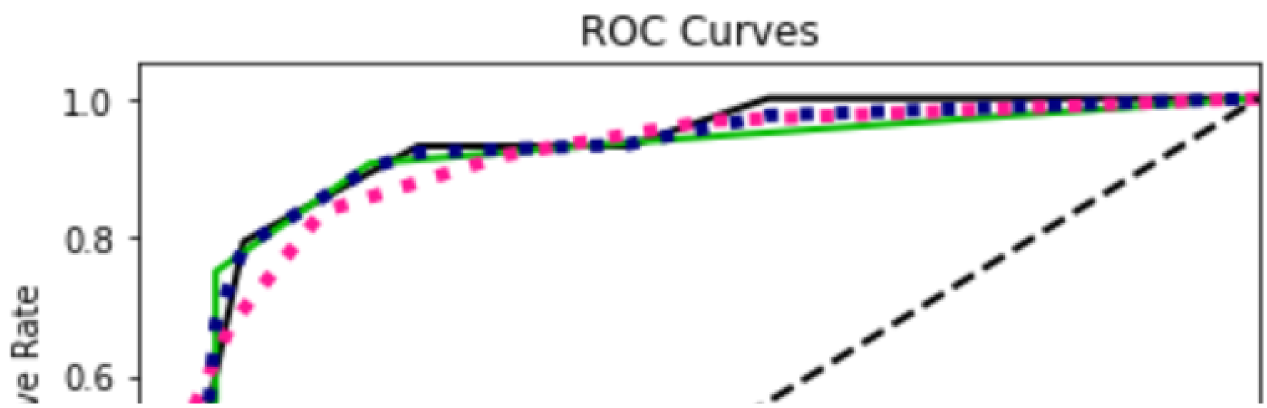


Image by Author

I note that the roc curve has improved. I try now with the over sampled training set. I omit the code because it is the same as before. In this case I obtain the best performance.



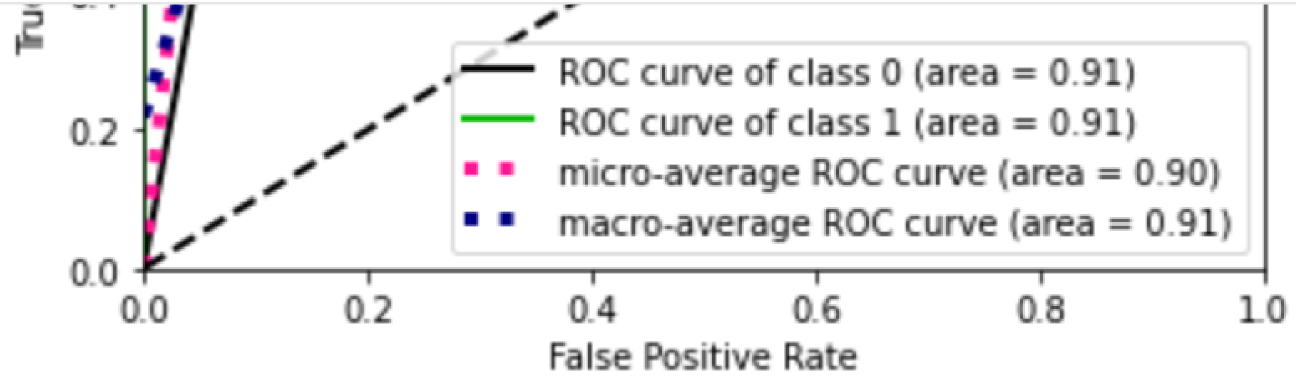


Image by Author

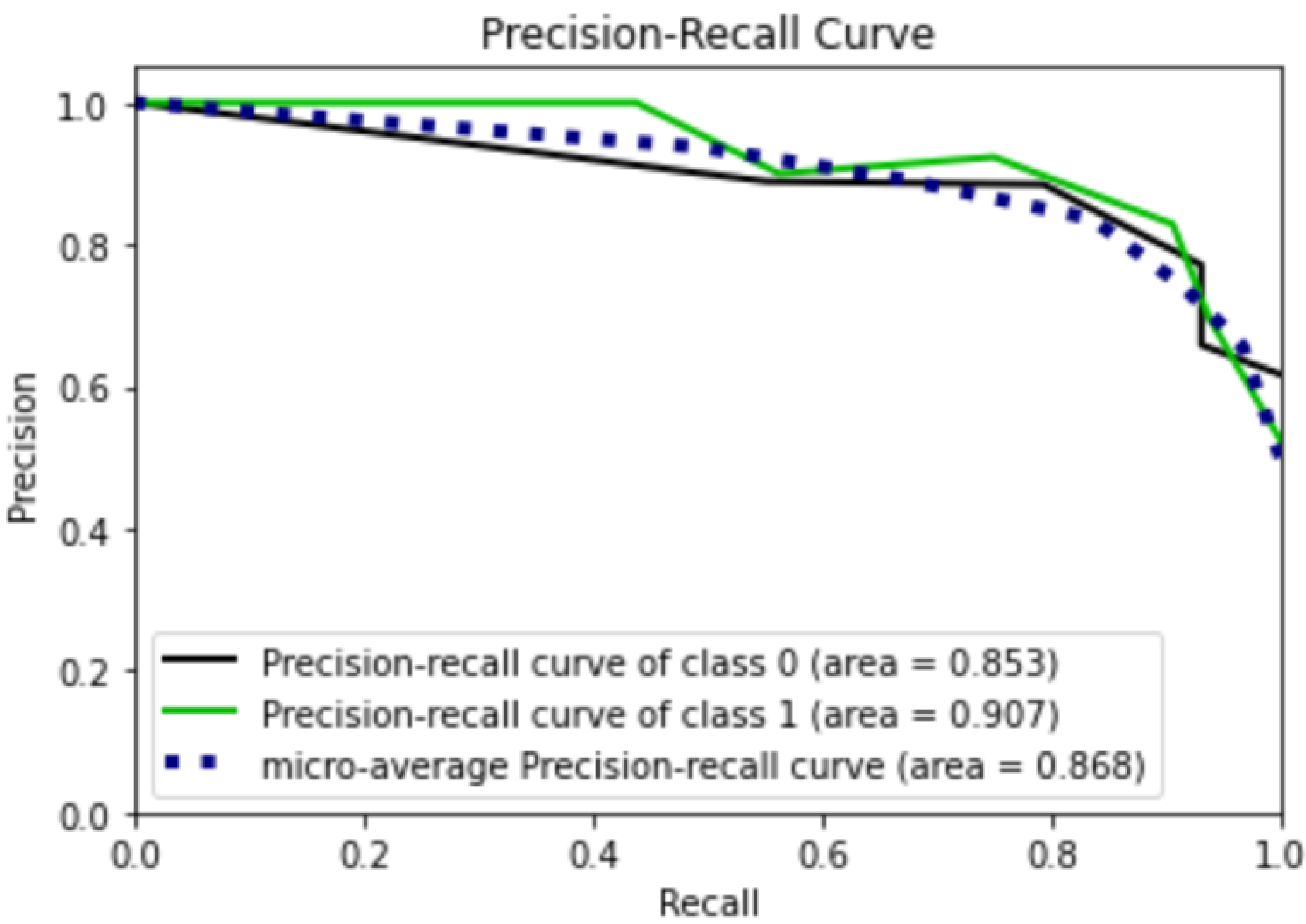


Image by Author

Summary

In this tutorial I have illustrated the full workflow to build a good model for data analysis. The workflow includes:

- data preprocessing, with features selection and balancing

- model evaluation, through the ROC curve and the Precision Recall curve.

In this tutorial I have not dealt with Outliers Detection. If you want to learn something about this aspect, you can give a look to my previous article.

If you wanted to be updated on my research and other activities, you can follow me on Twitter, Youtube and and Github.

Data Science Data Analysis Python Scikit Learn Imbalanced Data