Atelier 5 : Ensemble Learning (Boosting)

Pr. Ali Idri

Master of Science in Quantitative and Financial Modelling

2022-2023

Problem statement:

The Pima Indians dataset is a classic dataset in machine learning and is often used for classification tasks. It contains data on Pima Indian women aged 21 and older living near Phoenix, Arizona, USA. The goal of the classification task is to predict whether or not a woman will develop diabetes based on various medical measurements.

This task of classification is at the heart of medical diagnostic decision-support systems, which are becoming more widely used in the healthcare industry. Such machine-learning algorithms can greatly reduce the false negative rate (when patients are misdiagnosed until it is too late) and improve treatment and survival chances. It remains a difficult task due to subtle variations in patient data, making it hard for a **single model** to achieve **high performance**. Effective models can be constructed using ensemble learning that can improve the accuracy of the classification task by reducing overfitting, increasing the diversity of the models, and improving the generalization performance of the model.

Boosting is a an ensemble learning method used in machine learning to improve the models' predictive accuracy and reduce bias by converting multiple weak learners into a single strong learning model. There are three major boosting algorithms: Adaptive Boosting (AdaBoost), Gradient Boosting and Extreme Gradient Boosting (XGBoost).

Boosting creates an ensemble model by combining several weak decision trees sequentially. It assigns weights to the output of individual trees. Then it gives incorrect classifications from the first decision tree a higher weight and input to the next tree. After numerous cycles, the boosting method combines these weak rules into a single powerful prediction rule. This raises the question as to how many trees (weak learners or estimators) to configure in the boosting model.

The objective of this work is to design a systematic experiment to select the number of decision trees and the boosting method to use on our problem.

This case study will be tackled using:

- ✓ The scikit-learn library with Python.
- ✓ The boosting ensemble with different methods.
- ✓ The single machine-learning model "Decision Tree" to compare the performance of the boosting ensemble models.

This workshop tries to answer three research questions:

1- What is the number of trees that gives the best results for each boosting ensemble?

2- What is the best performing boosting method (AdaBoost, Gradient Boosting or XGBoost)?

3- Do the boosting ensembles outperform the Decision Tree model?

The examples of source code presented in this tutorial are proposals. At the end of the tutorial, you can make your own implementations and interpretations.

I. Loading the dataset:

Let's start by importing the libraries and dataset:

```
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
import matplotlib
from matplotlib import pyplot
import numpy
from sklearn.ensemble import AdaBoostClassifier
from sklearn.ensemble import GradientBoostingClassifier
from xgboost import XGBClassifier
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
```

We load the Pima Indians dataset and we create objects X and y to store the data and the target value respectively, and we randomly split the dataset into 70% for training and 30% for testing.

```
# Load the Pima Indians dataset
url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/pima-indians-diabetes.data.csv"
names = ['preg', 'plas', 'pres', 'skin', 'test', 'mass', 'pedi', 'age', 'class']
feature_names = names[:8]
data = pd.read_csv(url, names=names)

#Splitting the data into dependent and independent variables
```

```
X = data.drop("class", axis=1)
y=  data["class"]
# Splitting the data into training and testing data
X_train, X_test, y_train, y_test = train_test_split(X, y,train_size=0.7)
```

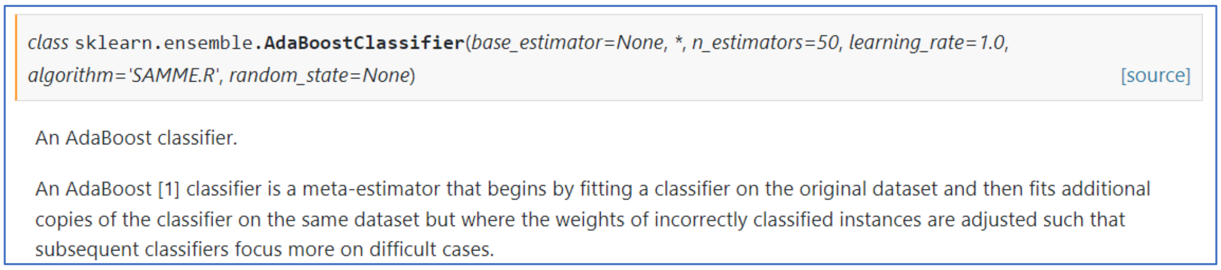
II. Training and evaluating the boosting ensembles:

In this section, we will train and evaluate variants of different boosting ensembles using different boosting methods: AdaBoost, Gradient Boosting and XGBoost with different number of trees (from 50 to 200).

1) AdaBoost:

For more details about the parameters of the Sklearn.Ensemble.AdaBoostClassifier, you can visit the page:

https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.AdaBoostClassifier.html



```
class sklearn.ensemble.AdaBoostClassifier(base_estimator=None, *, n_estimators=50, learning_rate=1.0,
algorithm='SAMME.R', random_state=None)                                                    [source]
```

An AdaBoost classifier.

An AdaBoost [1] classifier is a meta-estimator that begins by fitting a classifier on the original dataset and then fits additional copies of the classifier on the same dataset but where the weights of incorrectly classified instances are adjusted such that subsequent classifiers focus more on difficult cases.

Fig 2: AdaBoost Classifier on the sklearn.ensemble library

Let's start by implementing the AdaBoost ensemble using the default parameters:

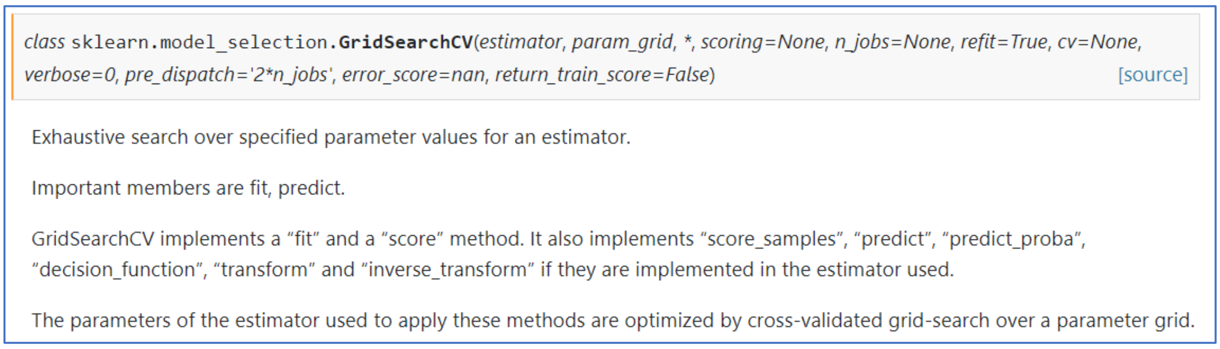
```
model_ada = AdaBoostClassifier()
model_ada.fit(X_train,y_train)
model_ada.score(X_test, y_test)
```

After training the ensemble on the training subset and testing it on the testing subset, we can calculate the accuracy of the Adaboost ensemble and we obtain: 94.74%.

It's possible to improve the accuracy score by using more trees, by default the number of trees is 50, so we train another Adaboost ensemble built with 100 trees by setting the parameter n_estimators to 100 and we calculate the score again:

```
model_ada = AdaBoostClassifier(n_estimators= 100)
model_ada.fit(X_train,y_train)
model_ada.score(X_test, y_test)
```

We obtain an accuracy score of: 95,90%. The Adaboost ensemble of 100 trees performs better than the one with 50 trees. To evaluate the effect of adding more decision trees to the boosting ensembles, we can perform a grid search with a series of values of trees. One of the tools available is Scikit-Learn’s GridSearchCV class. For more details about the sklearn.model_selection.GridSearchCV, you can visit the page:
https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html


class `sklearn.model_selection.`**`GridSearchCV`**(*estimator, param_grid, *, scoring=None, n_jobs=None, refit=True, cv=None, verbose=0, pre_dispatch='2*n_jobs', error_score=nan, return_train_score=False*) [source]

Exhaustive search over specified parameter values for an estimator.

Important members are fit, predict.

GridSearchCV implements a “fit” and a “score” method. It also implements “score_samples”, “predict”, “predict_proba”, “decision_function”, “transform” and “inverse_transform” if they are implemented in the estimator used.

The parameters of the estimator used to apply these methods are optimized by cross-validated grid-search over a parameter grid.

Before using GridSearchCV, let’s have a look on the important parameters.

- estimator: The model or function on which we want to use GridSearchCV
- param_grid: Dictionary or list of the estimator’s parameters in which GridSearchCV have to select the best.
- Scoring: The evaluation metric to use for the model to decide the best hyperparameters, if not specified then it uses estimator score.
- cv: The number of splits that is needed for cross validation (5 by default).
- n_jobs: The number of jobs to be run in parallel, -1 signifies to use all processor.

```
param_grid = dict(n_estimators= range(50,200,10))
grid_search_ada = GridSearchCV(model_ada, param_grid, scoring="accuracy", n_jobs=-1, cv=1
verbose=1)
grid_search_ada.fit(X, y)
means_ada = grid_search_ada.cv_results_['mean_test_score']
```

We make a dictionary called param_grid in which we specify the **n_estimators** parameter and the values to pass through GridSearchCV. We implement the AdaBoost method with different number of trees (from 50 to 200 with a step of 10) using the GridSearchCV and we plot the mean accuracy for each model to see how the number of trees affects the performance of the boosting ensemble (We define a function since we are going to plot the results for the other methods too):

```
n_estimators=range(50,200,10)
def plot_results(model,means):
  scores = numpy.array(means).reshape(len(n_estimators))
  pyplot.plot(n_estimators, scores, label=str(model)[:-2] )
  pyplot.legend()
  pyplot.xlabel('n_estimators')
  pyplot.ylabel('accuracy')
plot_results(model_ada, means_ada)
```

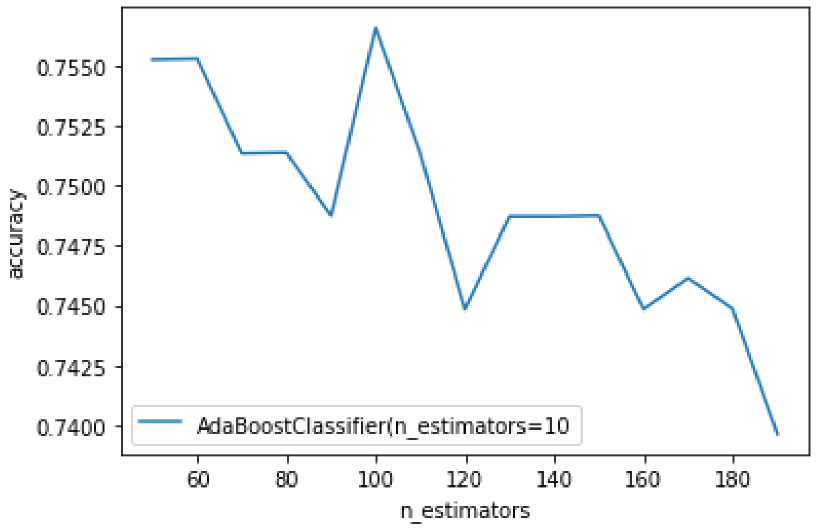
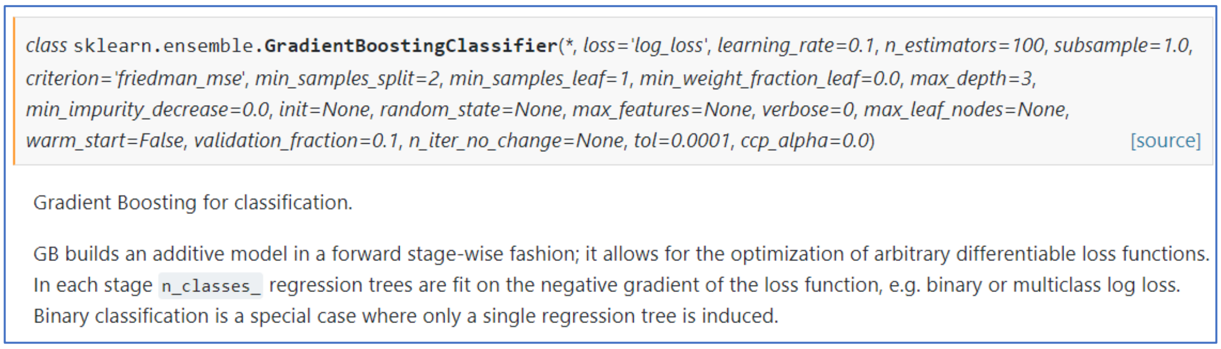


Fig 3: The accuracy of AdaBoost ensembles using different number of trees

2) Gradient Boosting:

For more details about the parameters of the Sklearn.Ensemble.GradientBoostingClassifier, you can visit the page:

https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html



class `sklearn.ensemble.`**`GradientBoostingClassifier`**(**, loss='log_loss', learning_rate=0.1, n_estimators=100, subsample=1.0, criterion='friedman_mse', min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_depth=3, min_impurity_decrease=0.0, init=None, random_state=None, max_features=None, verbose=0, max_leaf_nodes=None, warm_start=False, validation_fraction=0.1, n_iter_no_change=None, tol=0.0001, ccp_alpha=0.0*) [source]

Gradient Boosting for classification.

GB builds an additive model in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions. In each stage `n_classes_` regression trees are fit on the negative gradient of the loss function, e.g. binary or multiclass log loss. Binary classification is a special case where only a single regression tree is induced.

We implement the Gradient Boosting ensemble using the default parameters:

```
model_gbm = GradientBoostingClassifier()
model_gbm.fit(X_train,y_train)
model_gbm.score(X_test, y_test)
```

As we did before, we perform a grid search for Gradient Boosting, to evaluate the ensembles with different number of trees, and we plot the results:

```
grid_search_gbm = GridSearchCV(model_gbm, param_grid, scoring="accuracy", n_jobs=-1, cv=1
verbose=1)
grid_search_gbm.fit(X, y)
means_gbm = grid_search_gbm.cv_results_['mean_test_score']
plot_results(model_gbm,means_gbm )
```

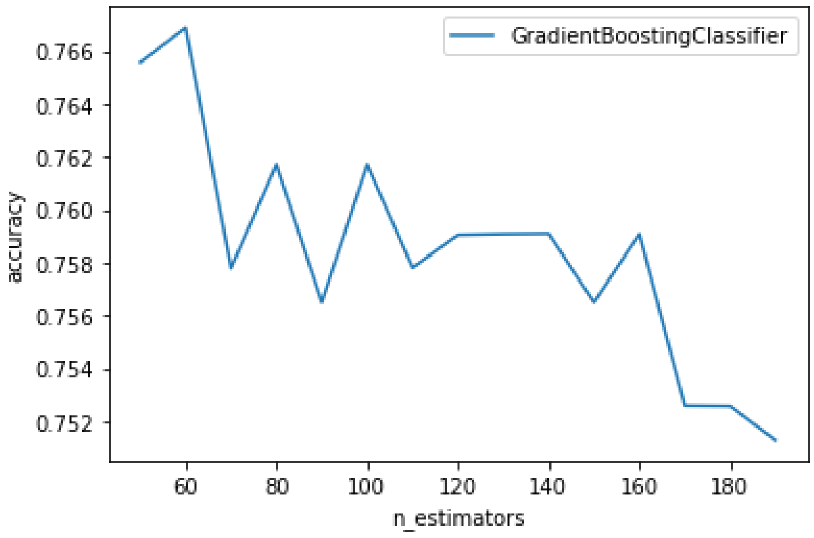
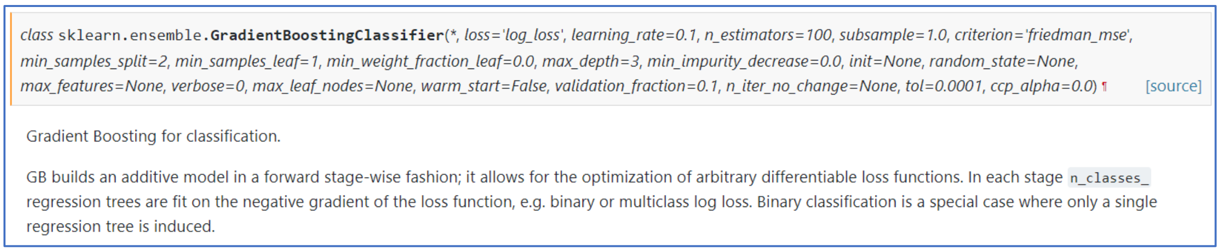


Fig 4: The accuracy of Gradient Boosting ensembles using different number of trees

3) XGBoost:

For more details about the parameters of the Sklearn.Ensemble.XGBClassifier, you can visit the page:

https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingClassifier.html



class sklearn.ensemble.**GradientBoostingClassifier**(**, loss='log_loss', learning_rate=0.1, n_estimators=100, subsample=1.0, criterion='friedman_mse', min_samples_split=2, min_samples_leaf=1, min_weight_fraction_leaf=0.0, max_depth=3, min_impurity_decrease=0.0, init=None, random_state=None, max_features=None, verbose=0, max_leaf_nodes=None, warm_start=False, validation_fraction=0.1, n_iter_no_change=None, tol=0.0001, ccp_alpha=0.0*) ¶ [source]

Gradient Boosting for classification.

GB builds an additive model in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions. In each stage `n_classes_` regression trees are fit on the negative gradient of the loss function, e.g. binary or multiclass log loss. Binary classification is a special case where only a single regression tree is induced.

We implement the XGboost ensemble using the default parameters:

```
model_xgb = XGBClassifier()
model_xgb.fit(X_train,y_train)
y_pred = model_xgb.predict(X_test)
model_xgb.score(X_test, y_test)
```

As we did before, we perform a grid search for XGBoost, to evaluate the ensembles with different number of trees, and we plot the results:

```
grid_search_xgb = GridSearchCV(model_ada, param_grid, scoring="accuracy", n_jobs=-1, cv=1
verbose=1)
grid_search_xgb.fit(X, y)
means_xgb = grid_search_xgb.cv_results_['mean_test_score']
plot_results(model_xgb,means_xgb)
```

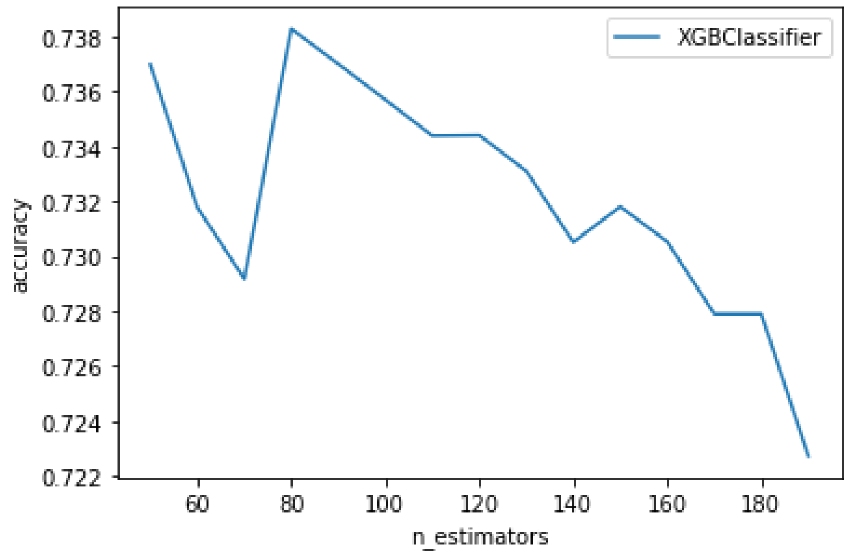


Fig 5: The accuracy of XGBoost ensembles using different number of trees

III. Comparing the different boosting methods:

To compare the different methods, we plot the accuracy of the ensembles using different number of trees for each boosting method in one figure.

```
models = ['AdaBoost','GBM','XGB']
means=numpy.concatenate((means_ada,means_gbm, means_xgb), axis=None)
scores = numpy.array(means).reshape(len(models), len(n_estimators))
for i, value in enumerate(models):
   pyplot.plot(n_estimators, scores[i], label=str(value))
pyplot.legend()
pyplot.xlabel('n_estimators')
pyplot.ylabel('accuracy')
pyplot.savefig('boosting_accuracy.png')
```

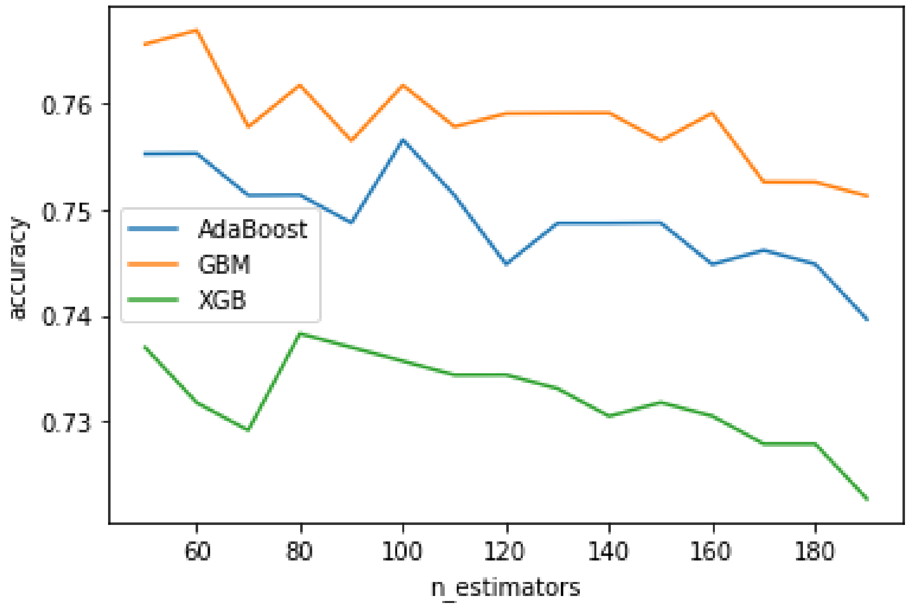


Fig 6: The accuracy of the boosting ensembles using different number of trees

We also compare the different methods in terms of the training time. We plot the mean training time of the ensembles using different number of trees for each boosting method to investigate the relationship between the number of trees and training time on our problem.

```
import numpy
models = ['AdaBoost','GBM','XGB']
means=numpy.concatenate((grid_search_ada.cv_results_['mean_score_time'],grid_search_gbm.cv_r
sults_['mean_score_time'],grid_search_xgb.cv_results_['mean_score_time']), axis=None)
scores = numpy.array(means).reshape(len(models), len(n_estimators))
for i, value in enumerate(models):
   pyplot.plot(n_estimators, scores[i], label=str(value))
pyplot.legend()
pyplot.xlabel('n_estimators')
pyplot.ylabel('training_time')
pyplot.savefig('boosting_time.png')
```

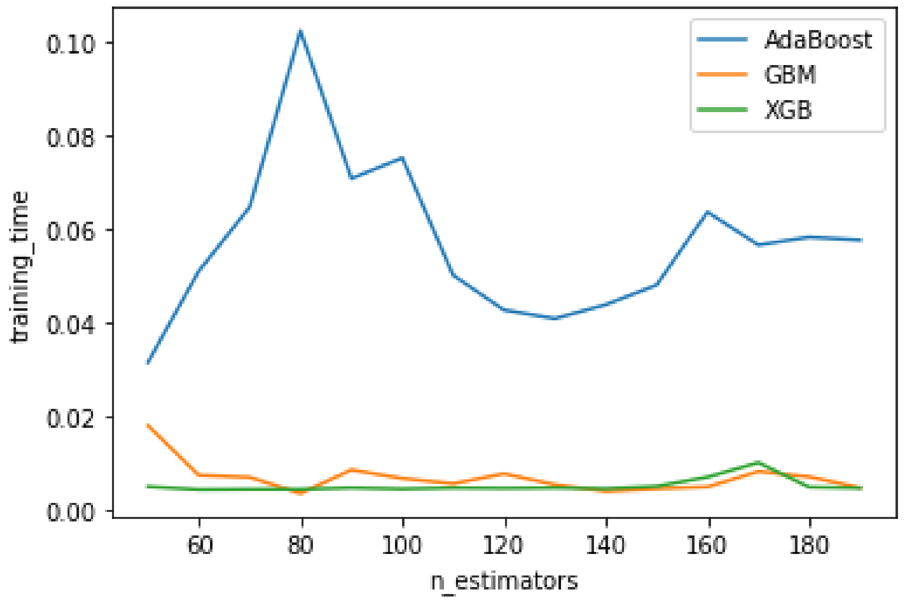


Fig 7: The training time of the boosting ensembles using different number of trees

We can print the results obtained by the grid search to identify the best number of trees for each boosting method (the number of trees that construct the ensemble with the highest accuracy) and the time taken by these ensembles to be trained.

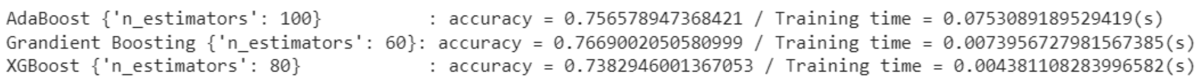
```
print("AdaBoost " +str(grid_search_ada.best_params_)+ "\t\t: accuracy =
"+str(grid_search_ada.best_score_)+" / Training time = "+
str(grid_search_ada.cv_results_['mean_score_time'][grid_search_ada.best_index_])+"(s)")

print("Grandient Boosting " +str(grid_search_gbm.best_params_)+ ": accuracy =
"+str(grid_search_gbm.best_score_)+" / Training time = "+
str(grid_search_gbm.cv_results_['mean_score_time'][grid_search_gbm.best_index_])+"(s)")

print("XGBoost "+str(grid_search_xgb.best_params_)+ "\t\t: accuracy =
"+str(grid_search_xgb.best_score_)+" / Training time = "+
str(grid_search_xgb.cv_results_['mean_score_time'][grid_search_xgb.best_index_])+"(s)")
```

The attribute **best_score_** return the mean cross-validated score of the best model (boosting ensemble) and the attribute **best_params_** returns the parameter setting that gave the best results on the hold out data.

The results:



```
AdaBoost {'n_estimators': 100}          : accuracy = 0.756578947368421 / Training time = 0.0753089189529419(s)
Grandient Boosting {'n_estimators': 60}: accuracy = 0.7669002050580999 / Training time = 0.0073956727981567385(s)
XGBoost {'n_estimators': 80}            : accuracy = 0.7382946001367053 / Training time = 0.004381108283996582(s)
```

Fig 8: the accuracy and training time of the best boosting ensembles

IV. Comparing the boosting ensembles with decision tree classifier:

Finally, we train a decision tree classifier and calculate its accuracy score, that will be compared with the accuracy scores of the boosting ensembles.

```
DT = DecisionTreeClassifier()
DT.fit(X_train,y_train)
DT.score(X_test,y_test)
```

Main findings:

- The accuracy of boosting ensembles improves when we add more trees.
- Adding more trees beyond a limit does not improve the performance of the boosting ensemble.
- XGboost is faster to train and performs better than Gradient Boosting and AdaBoost ensembles.

-The boosting ensembles outperform the decision tree classifier.

- You can use another Boosting method and compare it with the other ones (following the same steps)