In [1]:

```
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

%matplotlib inline

import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style("whitegrid")
sns.set_context("notebook")
#sns.set_context("poster")
```

In [2]:

```
# XGBoost is not included in the Anaconda distribution (yet... )
# Therefore you need to install it first
# ! pip install xgboost
# or
# ! sudo pip install xgboost
# or
# ! pip install --upgrade xgboost
# with sudo if you don't have admin privilegis
# in a Mac remember that you have to install Xcode and accept the license

from xgboost import XGBClassifier

from sklearn.model_selection import KFold
from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score

from sklearn.metrics import accuracy_score

from sklearn import preprocessing

import xgboost as xgb
```

XGBoost

XGBoost stands for eXtreme Gradient Boosting.

> The name XGBoost, though, actually refers to the engineering goal to push the limit of computations resources for boosted tree algorithms. Which is the reason why many people use XGBoost. Tianqui Chen, on Quora.com (tre creator of XGBoost)

XGBoosst is an implementation of Gradient Boosting Machines created by Tianqui Chen for his PhD thesis and now expanded with contributions from many developers.

If you are interested in the story of XGBoost, Tianqui Chen explains it in the tutorial *Story and Lessons Behind the Evolution of XGBoost*.

XGBoost needs to be downloaded and installed in your computer (if you have a Mac you need to download XCode - the Apple development suite - and accept the license agreement first). It has interfaces to many languages besides Python, such as R, Julia, C++, Scala, Java and JVM languages, etc ... XGBoost is distributed under the Apache-2 license.

XGBoost is built with a cloud platform focus in mind. In fact, AWS, Azure and Google Cloud host implementations of XGBoost tuned to their systems. Therefore XGBoost makes an extensive use of paralelization using all the cores and all the CPUs available and distributed computing for very large models. Also of techniques such out-of-core computeing for very large datasets and cache optimization to take advantage of the large cache memories in cloud servers.

Regarding the construction of the algorithm XGBoost is an implementation of gradient boosting. Gradient Boosting algorithms improve the solution building models that correct the errors of previous models. New models are created that predict the residuals or errors made by existing models. Models are added sequentially until no further improvement can be made. XGBoost supports the three main forms of gradient boosting:

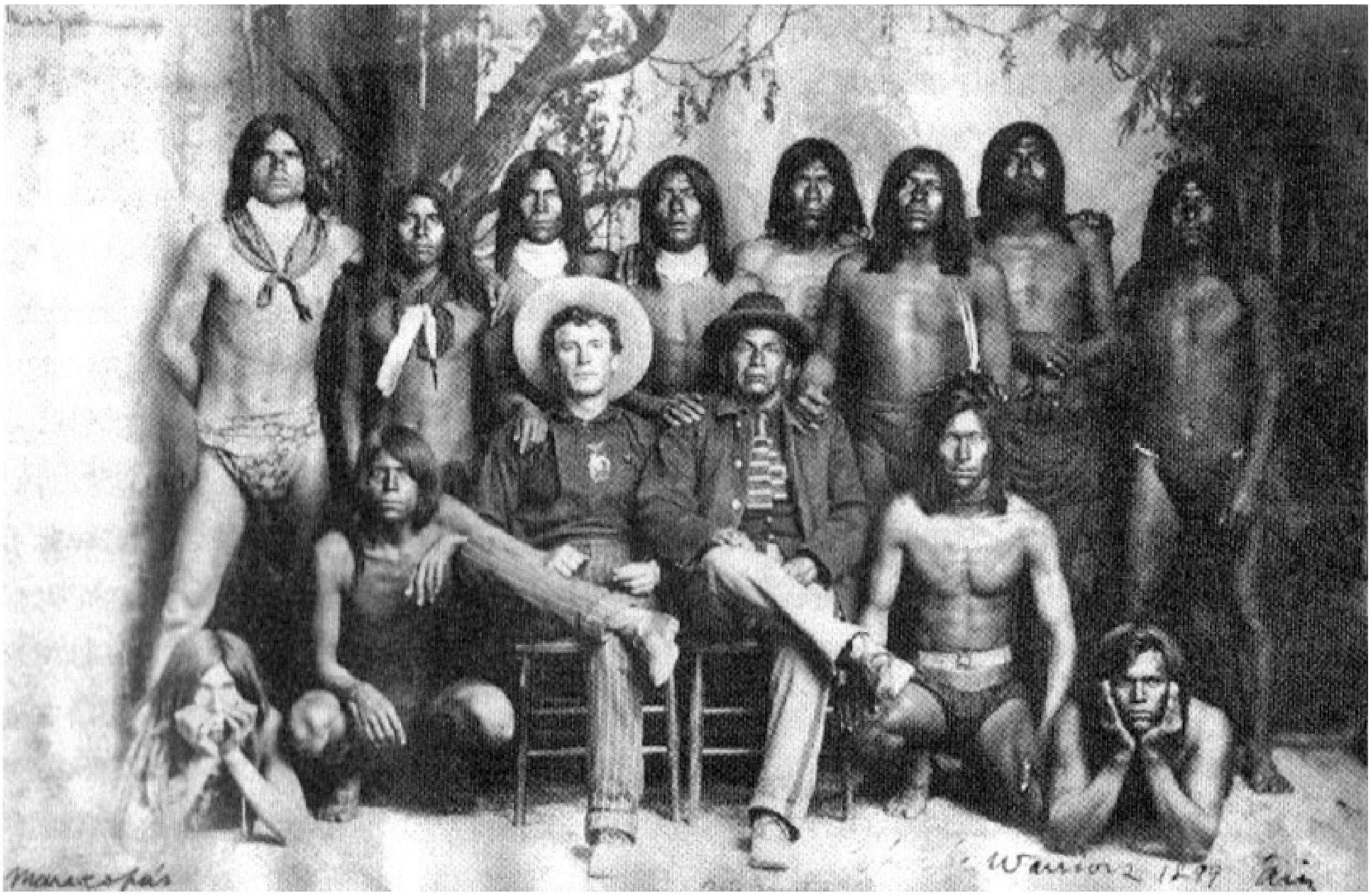
- **Gradient Boosting Algorithm (Gradient Boosting Machine Learning)**. Including the learning rate.
- **Sthocastic Gradient Boosting**. With sub-sampling at the row, column and column per split levels.
- **Regularized Gradient Boosting**. Using the L1 and L2 regularization (we've seen this in Ridge and Lasso regressions).

Two important additions are being sparse aware, therefore **automatically supporting missing values** and also supporting **continuous training so you can further boost an already fitted model with new data**.

One of the main reasons why XGBoost is so used is its efficiency, compared to other implementations of gradient boosting, it's fast, memory efficient and highly accurate (check *Benchmarking Random Forest Implementations* by Szilard Pafka).

So far XGBoost dominates the strutured or tabular datasets on classification and regression predictive modeling problems. It's the algorithm of choice for Kaggle competitions (*XGBoost: Machine Learning Challenge Winning Solutions*).

In order to be able to compare them with the previous one, we will use the same dataset, the Pima Indians.



In this exercise we will use one of the traditional Machine Learning dataset, the Pima Indians diabetes dataset.

This dataset is originally from the National Institute of Diabetes and Digestive and Kidney Diseases. The objective of the dataset is to diagnostically predict whether or not a patient has diabetes, based on certain diagnostic measurements included in the dataset. Several constraints were placed on the selection of these instances from a larger database. In particular, all patients here are females at least 21 years old of Pima Indian heritage.

Content The datasets consists of several medical predictor variables and one target variable, **Outcome**. Predictor variables includes the number of pregnancies the patient has had, their BMI, insulin level, age, and so on.

> - Pregnancies
> - Glucose
> - BloodPressure
> - SkinThickness
> - Insulin
> - BMI
> - DiabetesPedigreeFunction (scores de likelihood of diabetes based on family history)
> - Age
> - Outcome

In [3]:

```
# Load the Pima indians dataset and separate input and output components

from numpy import set_printoptions
set_printoptions(precision=3)

filename="pima-indians-diabetes.data.csv"
names=["pregnancies", "glucose", "pressure", "skin", "insulin", "bmi", "pedi", "age", "outc
p_indians=pd.read_csv(filename, names=names)
p_indians.head()

# First we separate into input and output components
array=p_indians.values
X=array[:,0:8]
y=array[:,8]
np.set_printoptions(suppress=True)
X
pd.DataFrame(X).head()
```

Out[3]:

	pregnancies	glucose	pressure	skin	insulin	bmi	pedi	age	outcome
0	6	148	72	35	0	33.6	0.627	50	1
1	1	85	66	29	0	26.6	0.351	31	0
2	8	183	64	0	0	23.3	0.672	32	1
3	1	89	66	23	94	28.1	0.167	21	0
4	0	137	40	35	168	43.1	2.288	33	1

Out[3]:

```
array([[  6.   , 148.   ,  72.   , ...,  33.6  ,   0.627,  50.   ],
       [  1.   ,  85.   ,  66.   , ...,  26.6  ,   0.351,  31.   ],
       [  8.   , 183.   ,  64.   , ...,  23.3  ,   0.672,  32.   ],
       ...,
       [  5.   , 121.   ,  72.   , ...,  26.2  ,   0.245,  30.   ],
       [  1.   , 126.   ,  60.   , ...,  30.1  ,   0.349,  47.   ],
       [  1.   ,  93.   ,  70.   , ...,  30.4  ,   0.315,  23.   ]])
```

Out[3]:

	0	1	2	3	4	5	6	7
0	6.0	148.0	72.0	35.0	0.0	33.6	0.627	50.0
1	1.0	85.0	66.0	29.0	0.0	26.6	0.351	31.0
2	8.0	183.0	64.0	0.0	0.0	23.3	0.672	32.0
3	1.0	89.0	66.0	23.0	94.0	28.1	0.167	21.0
4	0.0	137.0	40.0	35.0	168.0	43.1	2.288	33.0

XGBoost

XGBoost provides a wrapper to allow models to be treated like classifiers or regressors following the sckit-learn framework.

This means that you can use it in the same way that we use any other scikit-learn model.

For classification we will use the **XGBClassifier** class.

In [4]:

```
# XGBoost
#  evaluated with train & test - remember we have a high variance !

seed=7
test_size=0.4

# split into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size, random_state

# instantied the model
model=xgb.XGBClassifier()

# train the model on training data
model.fit(X_train, y_train)

# make predictions using test data
y_predict=model.predict(X_test)

# evaluate the predictions
accuracy = accuracy_score(y_test, y_predict)

print(f'XGBoost - Accuracy {accuracy*100:.3f}%')
```

Out[4]:

```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0,
              learning_rate=0.1, max_delta_step=0, max_depth=3,
              min_child_weight=1, missing=None, n_estimators=100, n_jobs=1,
              nthread=None, objective='binary:logistic', random_state=0,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
              silent=None, subsample=1, verbosity=1)


XGBoost - Accuracy 78.571%
```

In [5]:

```
# XGBoost
#   evaluated with KFold
#   in this case we use 3 splits because the amount of data is not large

seed=7

kfold=KFold(n_splits=3, random_state=seed, shuffle = True)

#learner=DecisionTreeClassifier(class_weight="balanced", random_state=seed)
learner=xgb.XGBClassifier()

results=cross_val_score(model, X, y, cv=kfold)

print(f'XGBoost with kfold - Accuracy {results.mean()*100:.3f}% std {results.std()*100:3f}'
```

```
XGBoost with kfold - Accuracy 76.562% std 1.940060
```

In [6]:

```
# XGBoost
#   evaluated with StratifiedKFold because of unbalanced classes
#   in this case we use 3 splits because the amount of data is not large

seed=7

kfold=StratifiedKFold(n_splits=3, random_state=seed, shuffle = True)

learner=xgb.XGBClassifier()

results=cross_val_score(model, X, y, cv=kfold)

print(f'XGBoost with Stratifiedkfold - Accuracy {results.mean()*100:.3f}% std {results.std(
```

```
XGBoost with Stratifiedkfold - Accuracy 76.693% std 1.756606
```

Plot a single XGBoost Decision Tree

Explainability of the algorithms is many times crucial. In this regard any tree based algorithm has a significant advantage because a tree representing the underlying structure of decisions can be plotted.

Requires the **graphviz** library installed.

In [7]:

```python
# Plotting a tree

# ! pip install graphviz

from matplotlib.pylab import rcParams

##set up the parameters
rcParams['figure.figsize'] = 150,150


model=XGBClassifier()

model.fit(X,y)

#plotting the first tree
xgb.plot_tree(model)

#plotting the fourth tree
xgb.plot_tree(model, num_trees=4)

#plotting from left to right
xgb.plot_tree(model, num_trees=4, rankdir="LR")

#fig = plt.gcf()
#fig.set_size_inches(150, 150)
#fig.savefig('xgb_tree.png')
```

Out[7]:

```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0,
              learning_rate=0.1, max_delta_step=0, max_depth=3,
              min_child_weight=1, missing=None, n_estimators=100, n_jobs=1,
              nthread=None, objective='binary:logistic', random_state=0,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
              silent=None, subsample=1, verbosity=1)
```

Out[7]:

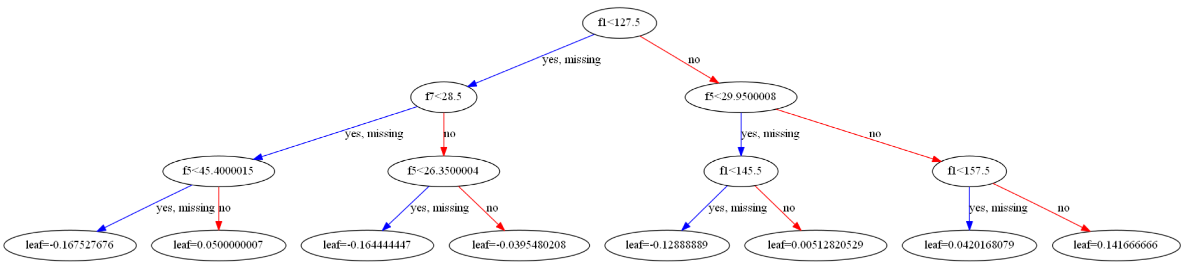
```
<matplotlib.axes._subplots.AxesSubplot at 0x1cd09782848>
```

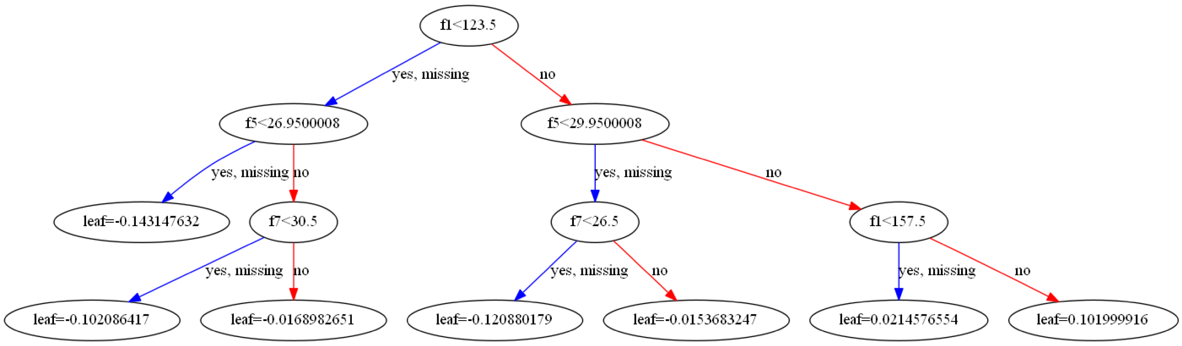
Out[7]:

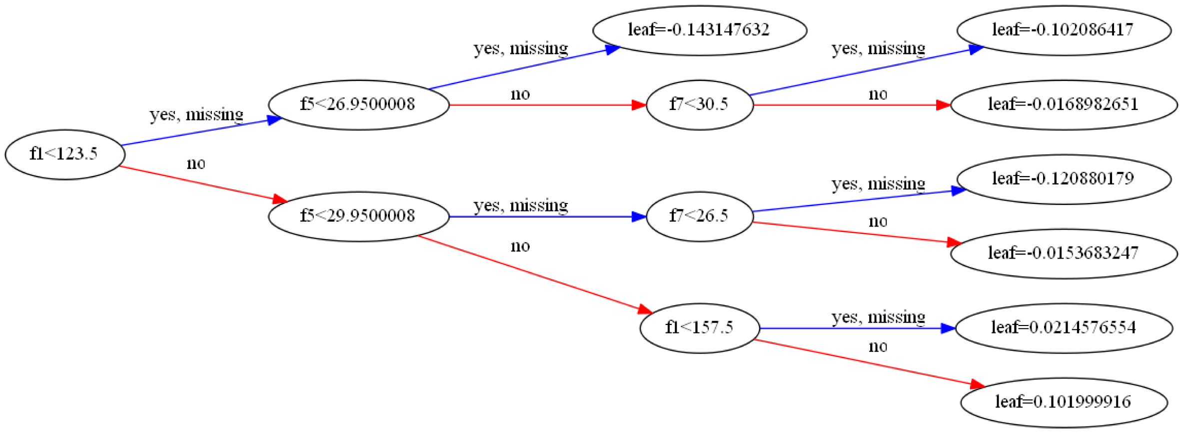
```
<matplotlib.axes._subplots.AxesSubplot at 0x1cd0b844048>
```

Out[7]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1cd0b883708>
```







Feature Importance

Similar to the Random Forest family we can obtain a vector with the relative importance of each feature and plot it.

In [8]:

```
# XGBoost - Feature importance


y_p=p_indians["outcome"]
X_p=p_indians.drop(["outcome"],axis=1)


# create a train/test split
#X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=7)
X_train, X_test, y_train, y_test = train_test_split(X_p, y_p, test_size=0.4, random_state=7

plt.figure(figsize=(15,9))

seed=7

model=XGBClassifier()
model.fit(X_train,y_train)

# Feature importance is calculated as the decrease in node impurity
# weighted by the probability of reaching that node.
# The node probability can be calculated by the number of samples that reach the node,
# divided by the total number of samples. The higher the value the more important the featu

for name, importance in zip(p_indians.columns, model.feature_importances_):
    print(f'{name:15s}  {importance:.4f}')

sns.barplot(x=p_indians.columns[:-1], y=model.feature_importances_)
```

Out[8]:

```
<Figure size 1080x648 with 0 Axes>
```

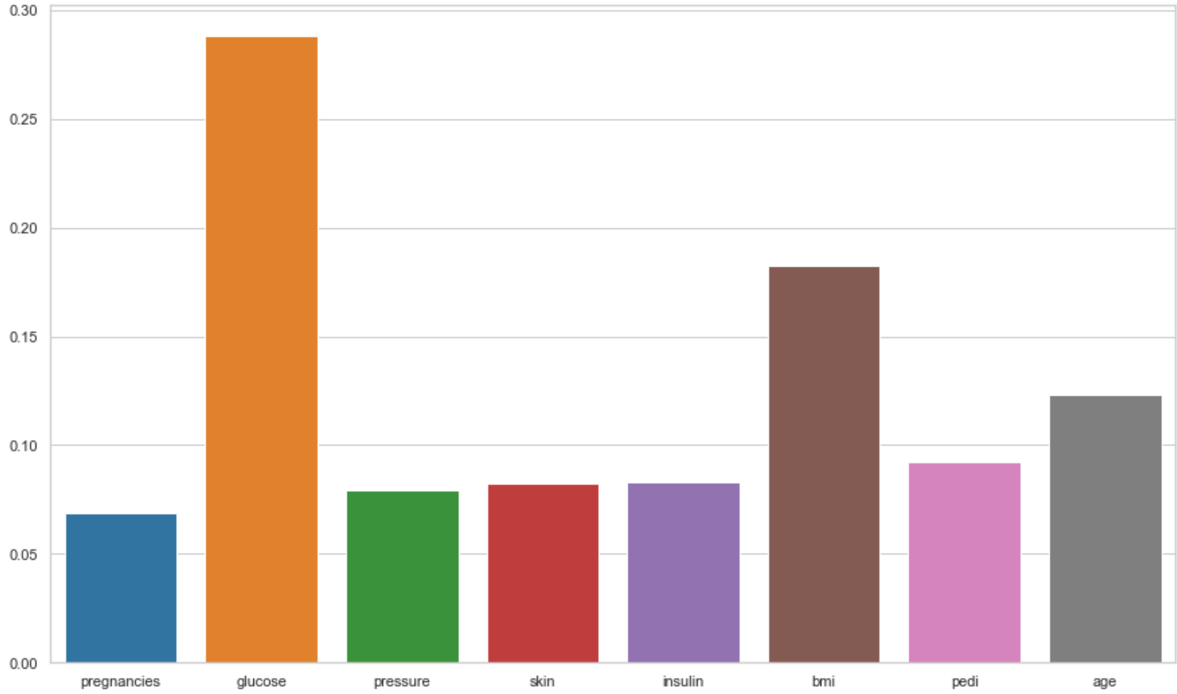
Out[8]:

```
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0,
              learning_rate=0.1, max_delta_step=0, max_depth=3,
              min_child_weight=1, missing=None, n_estimators=100, n_jobs=1,
              nthread=None, objective='binary:logistic', random_state=0,
              reg_alpha=0, reg_lambda=1, scale_pos_weight=1, seed=None,
              silent=None, subsample=1, verbosity=1)
```

```
pregnancies      0.0688
glucose          0.2879
pressure         0.0794
skin             0.0822
insulin          0.0832
bmi              0.1827
pedi             0.0925
age              0.1233
```

Out[8]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1cd78867788>
```


0.30
0.25
0.20
0.15
0.10
0.05
0.00
pregnancies
glucose
pressure
skin
insulin
bmi
pedi
age

In [9]:

```
import matplotlib.pylab as pl

y_p = p_indians["outcome"]
X_p = p_indians.drop(["outcome"],axis=1)


# create a train/test split
# X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=7)
X_train, X_test, y_train, y_test = train_test_split(X_p, y_p, test_size=0.4, random_state=7
d_train = xgb.DMatrix(X_train, label=y_train)
d_test = xgb.DMatrix(X_test, label=y_test)


params = {
    "eta": 0.01,
    "objective": "binary:logistic",
    "subsample": 0.5,
    "base_score": np.mean(y_train),
    "eval_metric": "logloss"
}
model = xgb.train(params, d_train, 5000, evals = [(d_test, "test")], verbose_eval=100, earl


# Weight. The number of times a feature is used to split the data across all trees.

ax = xgb.plot_importance(model, importance_type="weight")
pl.title("xgboost.plot_importance(model)")

print(f'Weight. The number of times a feature is used to split the data across all trees.')

ax.figure.set_size_inches(10,8)


# Cover. The number of times a feature is used to split the data across all trees
#         weighted by the number of training data points that go through those splits.

ax = xgb.plot_importance(model, importance_type="cover")
pl.title("xgboost.plot_importance(model, importance_type='cover')")

ax.figure.set_size_inches(10,8)


# Gain. The average training loss reduction gained when using a feature for splitting.

ax = xgb.plot_importance(model, importance_type="gain")
pl.title("xgboost.plot_importance(model, importance_type='gain')")

ax.figure.set_size_inches(10,8)
```

```
[0]     test-logloss:0.643018
Will train until test-logloss hasn't improved in 20 rounds.
[100]   test-logloss:0.506988

C:\Users\duart\AppData\Local\conda\conda\envs\testEnv\lib\site-packages\xgbo
ost\core.py:587: FutureWarning: Series.base is deprecated and will be remove
d in a future version
  if getattr(data, 'base', None) is not None and \
```

```
[200]	test-logloss:0.479592
Stopping. Best iteration:
[242]	test-logloss:0.475786
```

Out[9]:

```
Text(0.5, 1.0, 'xgboost.plot_importance(model)')
```

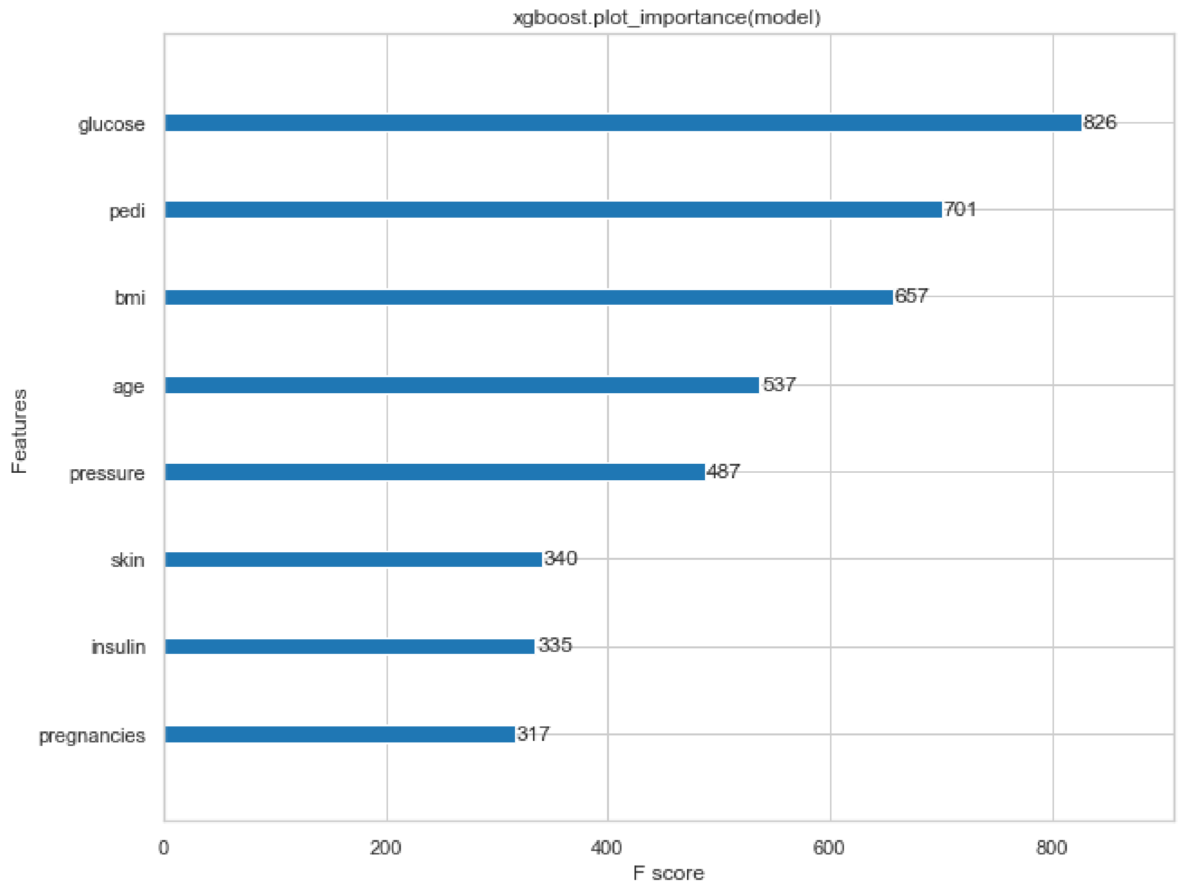
```
Weight. The number of times a feature is used to split the data across all t
rees.
```

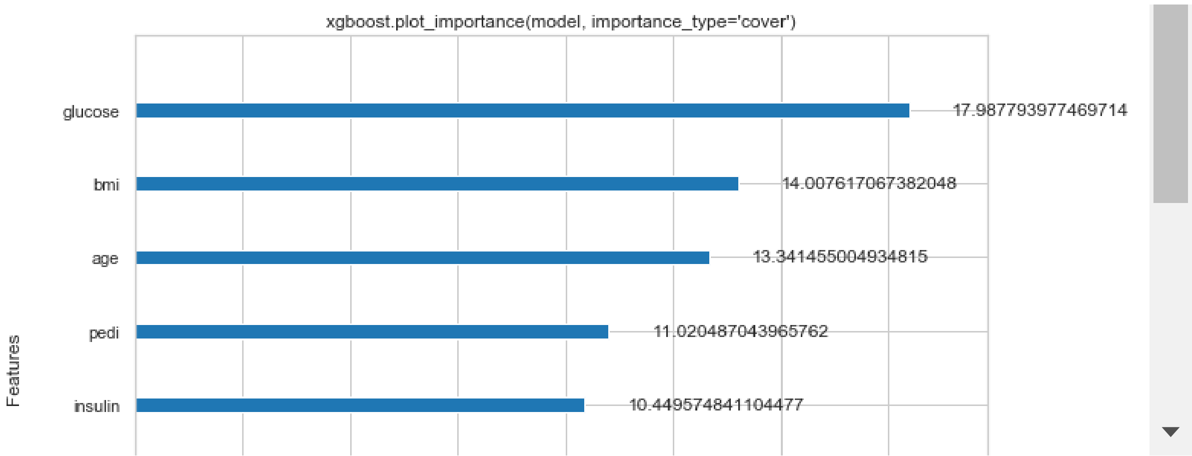
Out[9]:

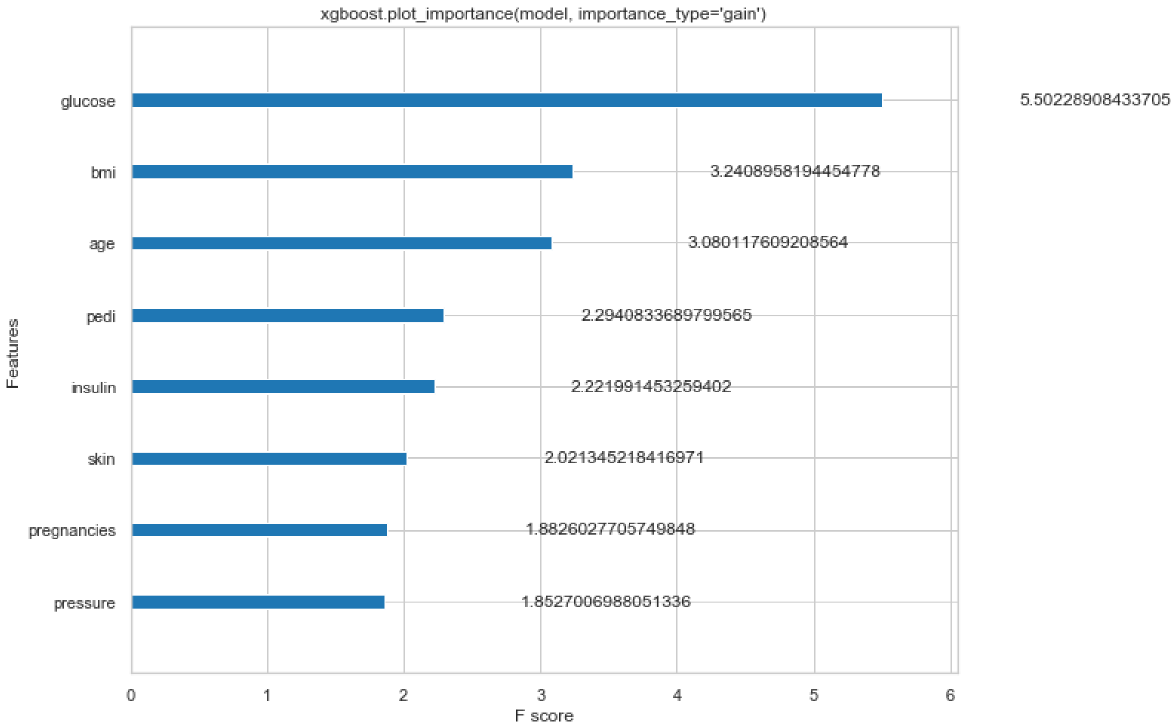
```
Text(0.5, 1.0, "xgboost.plot_importance(model, importance_type='cover')")
```

Out[9]:

```
Text(0.5, 1.0, "xgboost.plot_importance(model, importance_type='gain')")
```




xgboost.plot_importance(model, importance_type='cover')
Features
glucose
bmi
age
pedi
insulin
17.987793977469714
14.007617067382048
13.341455004934815
11.020487043965762
10.449574841104477


xgboost.plot_importance(model, importance_type='gain')
Features
glucose
bmi
age
pedi
insulin
skin
pregnancies
pressure
5.50228908433705
3.2408958194454778
3.080117609208564
2.2940833689799565
2.221991453259402
2.021345218416971
1.8826027705749848
1.8527006988051336
0
1
2
3
4
5
6
F score

Tuning XGBoost

As any other Ensemble algorith, XGBoost and in general Gradient Boosting machines can be tuned, improving their performance subtantially.

However, it has been an intense effort in this algorithms in order to have highly accurate default parameters. Therefore, you may think twice before engaging into tuning. Again, as any ensemble algorithm hyperparameter tuning can be expensive in terms of computational resources and you think of using a cloud platfrom for it.

In its simplest version, there are three main parameters that you might want to explore:

> - The number of Decision Trees. The default is a conservative 100.
> - The size of the Decision Trees. Default is 3.
> - The learning rate.

Tuning the number of Decision Trees

In [10]:

```
# XGBoost - Grid Search Parameter Tuning
#

from sklearn.model_selection import GridSearchCV

seed=7

model = XGBClassifier()

kfold=StratifiedKFold(n_splits=3, random_state=seed, shuffle = True)

param_grid={"n_estimators":[10, 25, 50, 100, 150, 200, 250, 300]}

grid=GridSearchCV(estimator=model, param_grid=param_grid, scoring="neg_log_loss", cv=kfold)
grid_result=grid.fit(X,y)

print(f'Grid Best Score {grid_result.best_score_:.7f} N. of estimators {grid_result.best_pa
print()

means = grid_result.cv_results_["mean_test_score"]
stds = grid_result.cv_results_["std_test_score"]
params = grid_result.cv_results_["params"]

for mean,std,param in zip(means,stds,params):
      print(f'N. estimators {param["n_estimators"]:3d} mean logloss {mean:.7f} ({std:.5f})'

plt.figure(figsize=(15,9))
sns.lineplot(x=param_grid["n_estimators"], y=means, linewidth=2.5)
plt.title("N. estimators & Logloss")
plt.xlabel("N. estimators")
plt.ylabel("Logloss")
```

```
Grid Best Score -0.4887563 N. of estimators  50

N. estimators  10 mean logloss -0.5366208 (0.00331)
N. estimators  25 mean logloss -0.4901138 (0.01166)
N. estimators  50 mean logloss -0.4887563 (0.01963)
N. estimators 100 mean logloss -0.5053880 (0.02124)
N. estimators 150 mean logloss -0.5195846 (0.02566)
N. estimators 200 mean logloss -0.5435639 (0.03365)
N. estimators 250 mean logloss -0.5641894 (0.03517)
N. estimators 300 mean logloss -0.5904812 (0.03639)
```

Out[10]:

```
<Figure size 1080x648 with 0 Axes>
```

Out[10]:

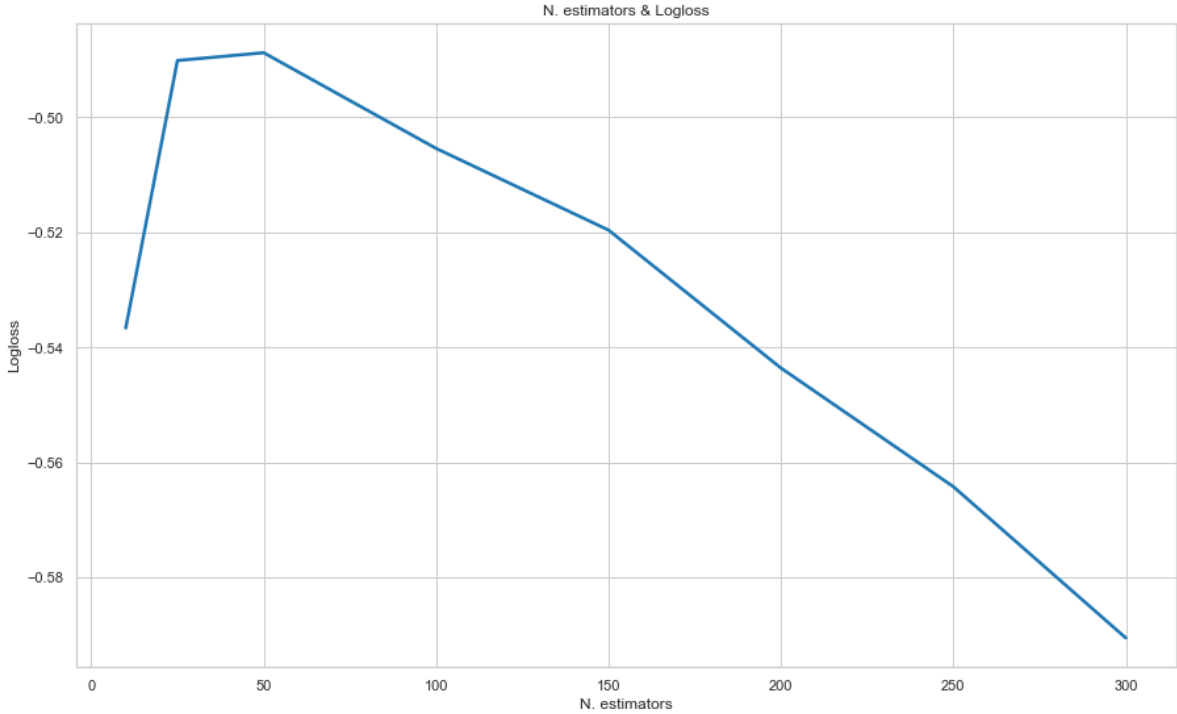
```
<matplotlib.axes._subplots.AxesSubplot at 0x1cd0c09b288>
```

Out[10]:

```
Text(0.5, 1.0, 'N. estimators & Logloss')
```

Out[10]:
```
Text(0.5, 0, 'N. estimators')
```

Out[10]:
```
Text(0, 0.5, 'Logloss')
```



Tuning the size of the Decision Trees

In [11]:

```
# XGBoost - Grid Search Parameter Tuning
#

from sklearn.model_selection import GridSearchCV

seed=7

model = XGBClassifier()

kfold=StratifiedKFold(n_splits=3, random_state=seed, shuffle = True)

param_grid={"max_depth":[1,2,3,4,5,6,7,8,9,10]}

grid=GridSearchCV(estimator=model, param_grid=param_grid, scoring="accuracy", cv=kfold)
grid_result=grid.fit(X,y)

print(f'Grid Best Score {grid_result.best_score_:.7f} Max Depth of Decision Trees {grid_res
print()

means = grid_result.cv_results_["mean_test_score"]
stds = grid_result.cv_results_["std_test_score"]
params = grid_result.cv_results_["params"]

for mean,std,param in zip(means,stds,params):
      print(f'N. estimators {param["max_depth"]:3d} depth of Decision Tree {mean:.7f} ({std

plt.figure(figsize=(15,9))
sns.lineplot(x=param_grid["max_depth"], y=means, linewidth=2.5)
plt.title("Depth of the Decision Trees & Accuracy")
plt.xlabel("Depth of the Decision Trees")
plt.ylabel("Accuracy")
```

```
Grid Best Score 0.7669271 Max Depth of Decision Trees   3

N. estimators   1 depth of Decision Tree 0.7591146 (0.01289)
N. estimators   2 depth of Decision Tree 0.7591146 (0.01814)
N. estimators   3 depth of Decision Tree 0.7669271 (0.01757)
N. estimators   4 depth of Decision Tree 0.7617188 (0.02532)
N. estimators   5 depth of Decision Tree 0.7539062 (0.03450)
N. estimators   6 depth of Decision Tree 0.7526042 (0.04051)
N. estimators   7 depth of Decision Tree 0.7408854 (0.03226)
N. estimators   8 depth of Decision Tree 0.7421875 (0.02780)
N. estimators   9 depth of Decision Tree 0.7369792 (0.03805)
N. estimators  10 depth of Decision Tree 0.7473958 (0.02859)
```

Out[11]:

```
<Figure size 1080x648 with 0 Axes>
```

Out[11]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1cd0bf25a88>
```

Out[11]:

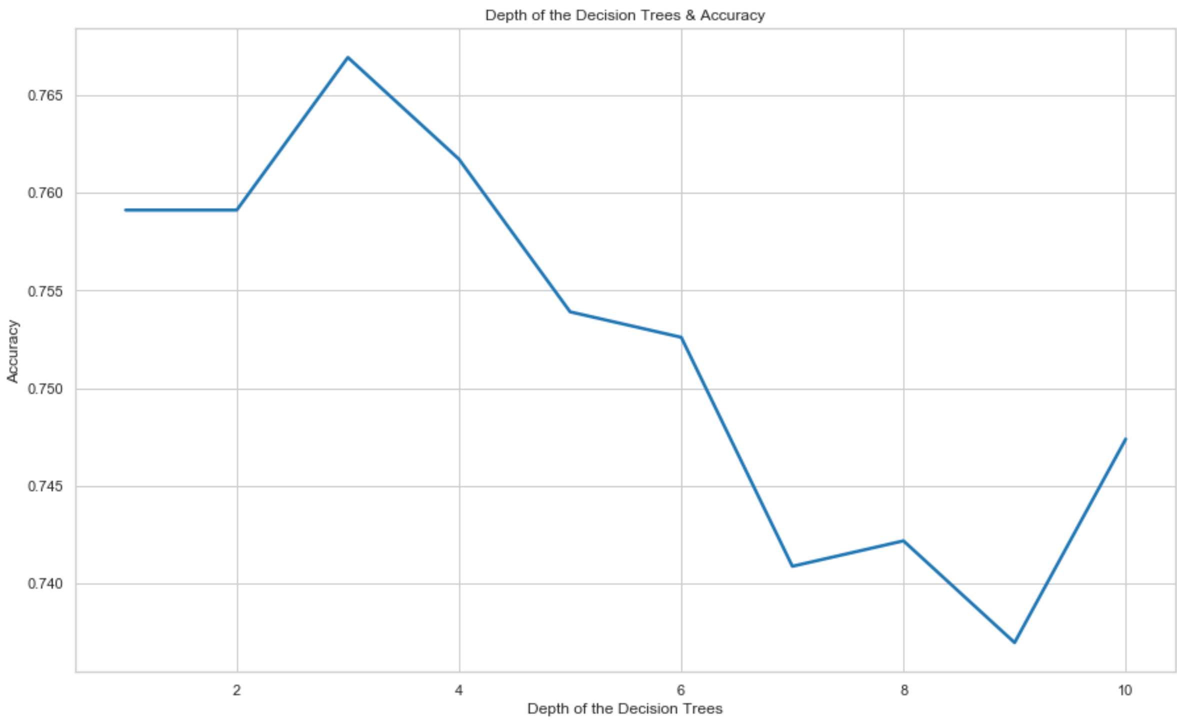
```
Text(0.5, 1.0, 'Depth of the Decision Trees & Accuracy')
```

Out[11]:

```
Text(0.5, 0, 'Depth of the Decision Trees')
```

Out[11]:

```
Text(0, 0.5, 'Accuracy')
```



Tuning the number & size of the Decision Trees

In [12]:

```
# XGBoost - Grid Search Parameter Tuning
#

from sklearn.model_selection import GridSearchCV

seed=7

model = XGBClassifier()

kfold=StratifiedKFold(n_splits=3, random_state=seed, shuffle = True)

param_grid={"max_depth":[1,2,3,4,5,6,7,8,9,10], "n_estimators":[50,100,150,200,250,300]}

grid=GridSearchCV(estimator=model, param_grid=param_grid, scoring="accuracy", cv=kfold)
grid_result=grid.fit(X,y)

print(f'Grid Best Score {grid_result.best_score_:.7f} \
      Number of trees {grid_result.best_params_["n_estimators"]:3d} \
      Max Depth of Decision Trees {grid_result.best_params_["max_depth"]:3d}')
print()

means = grid_result.cv_results_["mean_test_score"]
stds = grid_result.cv_results_["std_test_score"]
params = grid_result.cv_results_["params"]

tu_plot=pd.DataFrame(columns=["N estimators","Size of DT","accuracy"])
for mean,std,param in zip(means,stds,params):
    print(f'N. estimators {param["n_estimators"]:3d} \
            Depth {param["max_depth"]:3d} accuracy {mean:.7f} ({std:.5f})')
    tu_plot=tu_plot.append({"N estimators":param["n_estimators"],\
                            "Size of DT":param["max_depth"], "accuracy":mean}, ignore_index

plt.figure(figsize=(15,9))
sns.lineplot(data=tu_plot, x=tu_plot["N estimators"], y=tu_plot["accuracy"], \
             hue=tu_plot["Size of DT"], legend="full", palette=sns.color_palette("bright")
plt.title("N. Estimators & Depth of the Decision Trees")
plt.xlabel("N. Estimators")
plt.ylabel("accuracy")
```

```
Grid Best Score 0.7669271       Number of trees 100       Max Depth of Decis
ion Trees   3

N. estimators  50             Depth   1 accuracy 0.7526042 (0.00737)
N. estimators 100             Depth   1 accuracy 0.7591146 (0.01289)
N. estimators 150             Depth   1 accuracy 0.7578125 (0.01688)
N. estimators 200             Depth   1 accuracy 0.7565104 (0.01507)
N. estimators 250             Depth   1 accuracy 0.7591146 (0.01637)
N. estimators 300             Depth   1 accuracy 0.7565104 (0.01507)
N. estimators  50             Depth   2 accuracy 0.7591146 (0.01869)
N. estimators 100             Depth   2 accuracy 0.7591146 (0.01814)
N. estimators 150             Depth   2 accuracy 0.7565104 (0.02124)
N. estimators 200             Depth   2 accuracy 0.7578125 (0.01940)
N. estimators 250             Depth   2 accuracy 0.7578125 (0.01688)
N. estimators 300             Depth   2 accuracy 0.7552083 (0.01923)
N. estimators  50             Depth   3 accuracy 0.7591146 (0.02558)
N. estimators 100             Depth   3 accuracy 0.7669271 (0.01757)
N. estimators 150             Depth   3 accuracy 0.7617188 (0.01992)
```

```
N. estimators 200           Depth   3 accuracy 0.7643229 (0.03015)
N. estimators 250           Depth   3 accuracy 0.7552083 (0.03380)
N. estimators 300           Depth   3 accuracy 0.7500000 (0.02725)
N. estimators  50           Depth   4 accuracy 0.7630208 (0.02124)
N. estimators 100           Depth   4 accuracy 0.7617188 (0.02532)
N. estimators 150           Depth   4 accuracy 0.7526042 (0.02675)
N. estimators 200           Depth   4 accuracy 0.7408854 (0.03258)
N. estimators 250           Depth   4 accuracy 0.7421875 (0.03330)
N. estimators 300           Depth   4 accuracy 0.7343750 (0.03450)
N. estimators  50           Depth   5 accuracy 0.7630208 (0.02894)
N. estimators 100           Depth   5 accuracy 0.7539062 (0.03450)
N. estimators 150           Depth   5 accuracy 0.7473958 (0.02859)
N. estimators 200           Depth   5 accuracy 0.7500000 (0.03043)
N. estimators 250           Depth   5 accuracy 0.7434896 (0.03195)
N. estimators 300           Depth   5 accuracy 0.7421875 (0.03375)
N. estimators  50           Depth   6 accuracy 0.7473958 (0.03147)
N. estimators 100           Depth   6 accuracy 0.7526042 (0.04051)
N. estimators 150           Depth   6 accuracy 0.7447917 (0.04330)
N. estimators 200           Depth   6 accuracy 0.7500000 (0.03330)
N. estimators 250           Depth   6 accuracy 0.7500000 (0.02780)
N. estimators 300           Depth   6 accuracy 0.7460938 (0.03043)
N. estimators  50           Depth   7 accuracy 0.7486979 (0.03015)
N. estimators 100           Depth   7 accuracy 0.7408854 (0.03226)
N. estimators 150           Depth   7 accuracy 0.7369792 (0.03226)
N. estimators 200           Depth   7 accuracy 0.7408854 (0.03226)
N. estimators 250           Depth   7 accuracy 0.7421875 (0.02762)
N. estimators 300           Depth   7 accuracy 0.7369792 (0.02713)
N. estimators  50           Depth   8 accuracy 0.7486979 (0.03258)
N. estimators 100           Depth   8 accuracy 0.7421875 (0.02780)
N. estimators 150           Depth   8 accuracy 0.7369792 (0.03015)
N. estimators 200           Depth   8 accuracy 0.7382812 (0.03043)
N. estimators 250           Depth   8 accuracy 0.7408854 (0.03226)
N. estimators 300           Depth   8 accuracy 0.7421875 (0.03315)
N. estimators  50           Depth   9 accuracy 0.7434896 (0.03195)
N. estimators 100           Depth   9 accuracy 0.7369792 (0.03805)
N. estimators 150           Depth   9 accuracy 0.7408854 (0.03858)
N. estimators 200           Depth   9 accuracy 0.7421875 (0.03919)
N. estimators 250           Depth   9 accuracy 0.7356771 (0.03988)
N. estimators 300           Depth   9 accuracy 0.7356771 (0.04039)
N. estimators  50           Depth  10 accuracy 0.7473958 (0.03130)
N. estimators 100           Depth  10 accuracy 0.7473958 (0.02859)
N. estimators 150           Depth  10 accuracy 0.7486979 (0.02963)
N. estimators 200           Depth  10 accuracy 0.7460938 (0.03330)
N. estimators 250           Depth  10 accuracy 0.7447917 (0.03226)
N. estimators 300           Depth  10 accuracy 0.7434896 (0.03195)
```

Out[12]:

```
<Figure size 1080x648 with 0 Axes>
```

Out[12]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1cd0bf9d8c8>
```

Out[12]:

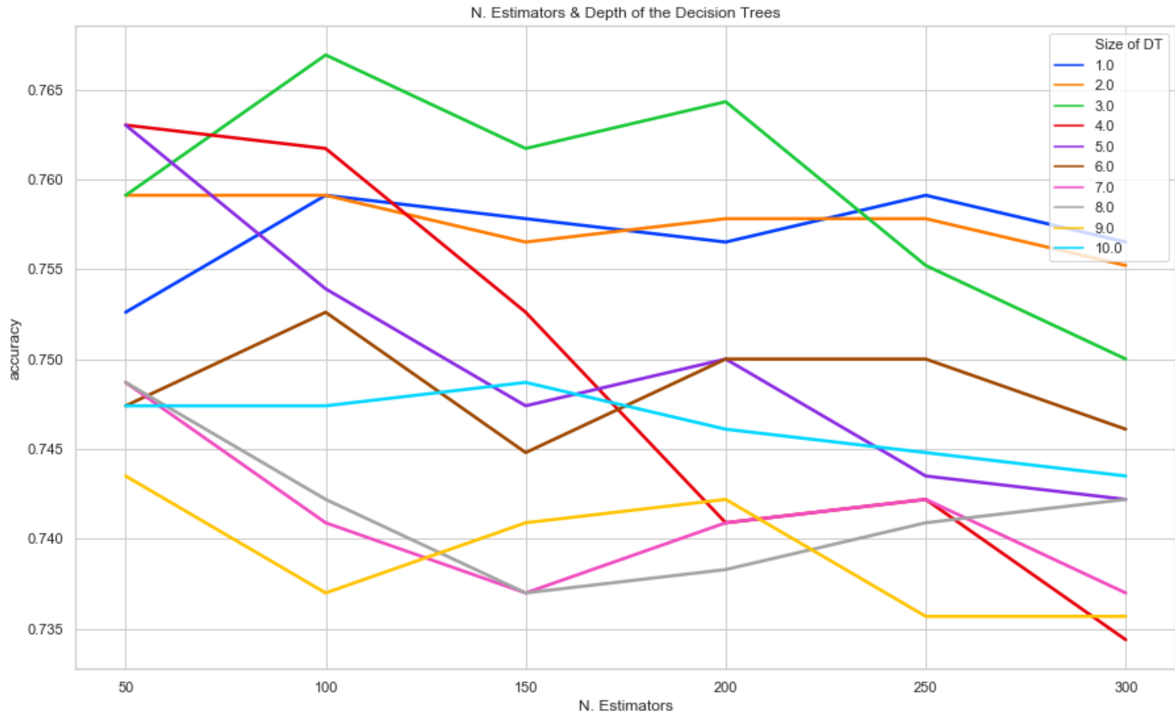
```
Text(0.5, 1.0, 'N. Estimators & Depth of the Decision Trees')
```

Out[12]:

```
Text(0.5, 0, 'N. Estimators')
```

Out[12]:

```
Text(0, 0.5, 'accuracy')
```



In [ ]:

In [ ]:

In [ ]:

In [ ]:

Mission 1

a) Use XGBoost with the Titanic dataset.

b) Discuss Feature importance, obtained with XGBoost, in the Titanic Dataset and relate it to the Victorian society.

In [13]:

```
# a) Use XGBoost with the Titanic dataset

# Load titanic dataset and create gender variable

titanic = pd.read_csv("titanic.csv")

titanic["Gender"] = titanic["Sex"].apply(lambda d: 1 if d == "female" else 0)

titanic.drop(["Name", "Sex"], axis = 1, inplace = True)

# Separate x and y

array = titanic.values
y = array[:,0]
x = array[0:,1:]
```

In [14]:

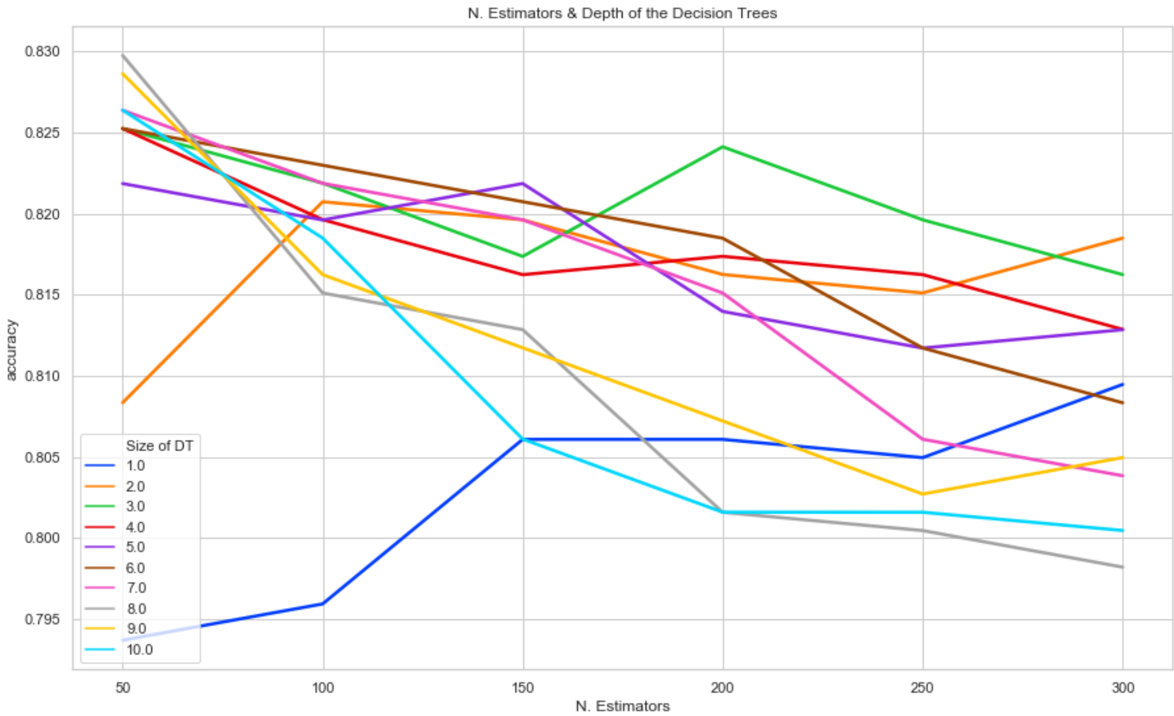
```
# Tune XGBoost

def tune(x,y):

    seed=7
    model = XGBClassifier()
    kfold=StratifiedKFold(n_splits=3, random_state=seed, shuffle = True)

    param_grid={"max_depth":[1,2,3,4,5,6,7,8,9,10], "n_estimators":[50,100,150,200,250,300]

    grid=GridSearchCV(estimator=model, param_grid=param_grid, scoring="accuracy", cv=kfold)
    grid_result=grid.fit(x,y)

    print(f'Grid Best Score {grid_result.best_score_:.7f} \
          Number of trees {grid_result.best_params_["n_estimators"]:3d} \
          Max Depth of Decision Trees {grid_result.best_params_["max_depth"]:3d}')
    print()

    means = grid_result.cv_results_["mean_test_score"]
    stds = grid_result.cv_results_["std_test_score"]
    params = grid_result.cv_results_["params"]

    tu_plot=pd.DataFrame(columns=["N estimators","Size of DT","accuracy"])
    for mean,std,param in zip(means,stds,params):
        print(f'N. estimators {param["n_estimators"]:3d} \
                Depth {param["max_depth"]:3d} accuracy {mean:.7f} ({std:.5f})')
        tu_plot=tu_plot.append({"N estimators":param["n_estimators"],\
                                "Size of DT":param["max_depth"], "accuracy":mean}, ignore_i

    plt.figure(figsize=(15,9))
    sns.lineplot(data=tu_plot, x=tu_plot["N estimators"], y=tu_plot["accuracy"], \
                 hue=tu_plot["Size of DT"], legend="full", palette=sns.color_palette("brigh
    plt.title("N. Estimators & Depth of the Decision Trees")
    plt.xlabel("N. Estimators")
    plt.ylabel("accuracy")
```

In [15]:

```
tune(x,y)
```

```
Grid Best Score 0.8297565          Number of trees  50          Max Dept
h of Decision Trees   8

N. estimators  50              Depth   1 accuracy 0.7936899 (0.03179)
N. estimators 100              Depth   1 accuracy 0.7959307 (0.02307)
N. estimators 150              Depth   1 accuracy 0.8060773 (0.02571)
N. estimators 200              Depth   1 accuracy 0.8060773 (0.02306)
N. estimators 250              Depth   1 accuracy 0.8049511 (0.02418)
N. estimators 300              Depth   1 accuracy 0.8094671 (0.02215)
N. estimators  50              Depth   2 accuracy 0.8083410 (0.00177)
N. estimators 100              Depth   2 accuracy 0.8207322 (0.01942)
N. estimators 150              Depth   2 accuracy 0.8196175 (0.01242)
N. estimators 200              Depth   2 accuracy 0.8162468 (0.02078)
N. estimators 250              Depth   2 accuracy 0.8151092 (0.01935)
N. estimators 300              Depth   2 accuracy 0.8184876 (0.02072)
N. estimators  50              Depth   3 accuracy 0.8252443 (0.02417)
N. estimators 100              Depth   3 accuracy 0.8218774 (0.02102)
N. estimators 150              Depth   3 accuracy 0.8173614 (0.02483)
N. estimators 200              Depth   3 accuracy 0.8241220 (0.02360)
N. estimators 250              Depth   3 accuracy 0.8196175 (0.02069)
N. estimators 300              Depth   3 accuracy 0.8162391 (0.02499)
N. estimators  50              Depth   4 accuracy 0.8252481 (0.02780)
N. estimators 100              Depth   4 accuracy 0.8196290 (0.02337)
N. estimators 150              Depth   4 accuracy 0.8162353 (0.02345)
N. estimators 200              Depth   4 accuracy 0.8173652 (0.02079)
N. estimators 250              Depth   4 accuracy 0.8162391 (0.02774)
N. estimators 300              Depth   4 accuracy 0.8128607 (0.02404)
N. estimators  50              Depth   5 accuracy 0.8218545 (0.01854)
N. estimators 100              Depth   5 accuracy 0.8196099 (0.01843)
N. estimators 150              Depth   5 accuracy 0.8218545 (0.02086)
N. estimators 200              Depth   5 accuracy 0.8139678 (0.02461)
N. estimators 250              Depth   5 accuracy 0.8117079 (0.02087)
N. estimators 300              Depth   5 accuracy 0.8128417 (0.01785)
N. estimators  50              Depth   6 accuracy 0.8252481 (0.02234)
N. estimators 100              Depth   6 accuracy 0.8229844 (0.02009)
N. estimators 150              Depth   6 accuracy 0.8207360 (0.01996)
N. estimators 200              Depth   6 accuracy 0.8184837 (0.01157)
N. estimators 250              Depth   6 accuracy 0.8117117 (0.01380)
N. estimators 300              Depth   6 accuracy 0.8083333 (0.01374)
N. estimators  50              Depth   7 accuracy 0.8263781 (0.01392)
N. estimators 100              Depth   7 accuracy 0.8218698 (0.01521)
N. estimators 150              Depth   7 accuracy 0.8196175 (0.01518)
N. estimators 200              Depth   7 accuracy 0.8151015 (0.01157)
N. estimators 250              Depth   7 accuracy 0.8060849 (0.01120)
N. estimators 300              Depth   7 accuracy 0.8038326 (0.00553)
N. estimators  50              Depth   8 accuracy 0.8297565 (0.01312)
N. estimators 100              Depth   8 accuracy 0.8151054 (0.01521)
N. estimators 150              Depth   8 accuracy 0.8128455 (0.01056)
N. estimators 200              Depth   8 accuracy 0.8015804 (0.01111)
N. estimators 250              Depth   8 accuracy 0.8004543 (0.00463)
N. estimators 300              Depth   8 accuracy 0.7981982 (0.01111)
N. estimators  50              Depth   9 accuracy 0.8286303 (0.01157)
N. estimators 100              Depth   9 accuracy 0.8162353 (0.00967)
N. estimators 150              Depth   9 accuracy 0.8117270 (0.00567)
N. estimators 200              Depth   9 accuracy 0.8072148 (0.00553)
N. estimators 250              Depth   9 accuracy 0.8027065 (0.00691)
```

```
N. estimators 300              Depth   9 accuracy 0.8049550 (0.00899)
N. estimators  50              Depth  10 accuracy 0.8263819 (0.01518)
N. estimators 100              Depth  10 accuracy 0.8184914 (0.01111)
N. estimators 150              Depth  10 accuracy 0.8060964 (0.01349)
N. estimators 200              Depth  10 accuracy 0.8015880 (0.01259)
N. estimators 250              Depth  10 accuracy 0.8015842 (0.01295)
N. estimators 300              Depth  10 accuracy 0.8004581 (0.01982)
```



In [16]:

```
# We will use 200 estimators, 3 estimators

# XGBoost
#  evaluated with train & test - remember we have a high variance !

def run_xg_boost(x, y, estimators, depth):

    seed=7
    test_size=0.4

    # split into train and test
    X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=test_size, random_s

    # instantied the model
    model=xgb.XGBClassifier(max_depth = depth, n_estimators = estimators)

    # train the model on training data
    model.fit(X_train, y_train)

    # make predictions using test data
    y_predict=model.predict(X_test)

    # evaluate the predictions
    accuracy = accuracy_score(y_test, y_predict)

    print(f'XGBoost - Accuracy {accuracy*100:.3f}%')
```

In [17]:

```
# Picked 3 as the maximum depth and 200 minimum trees (higher accuracy with 50 trees, but i

run_xg_boost(x, y, 200, 3)
```

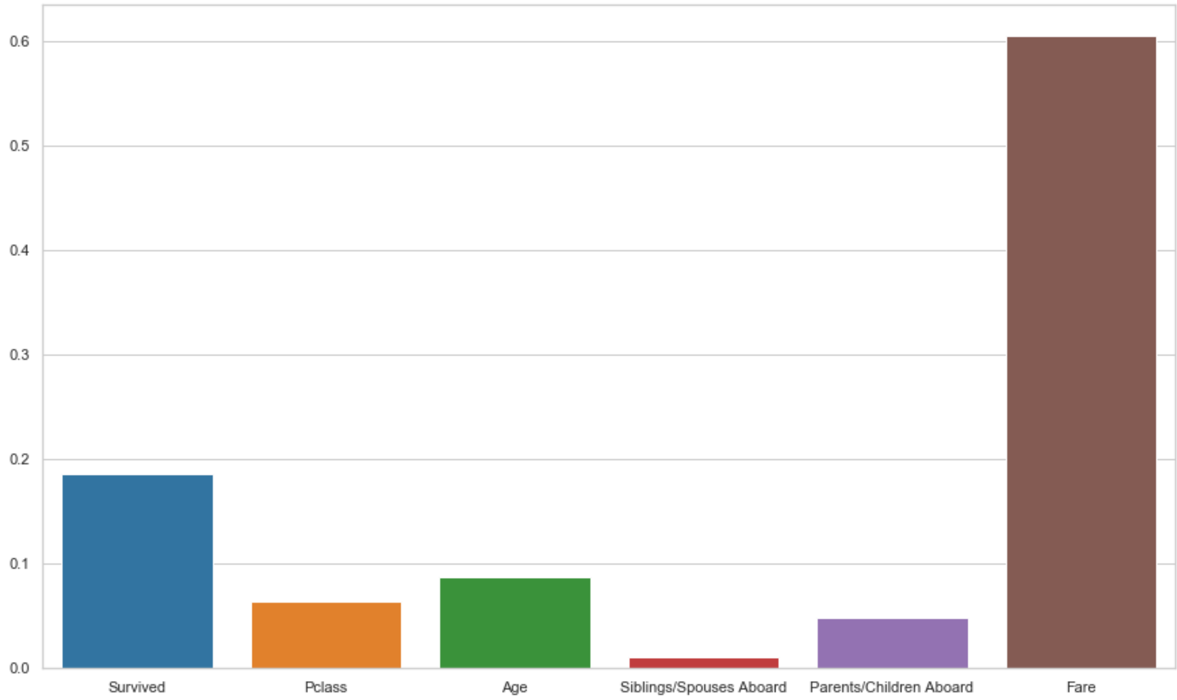
```
XGBoost - Accuracy 85.634%
```

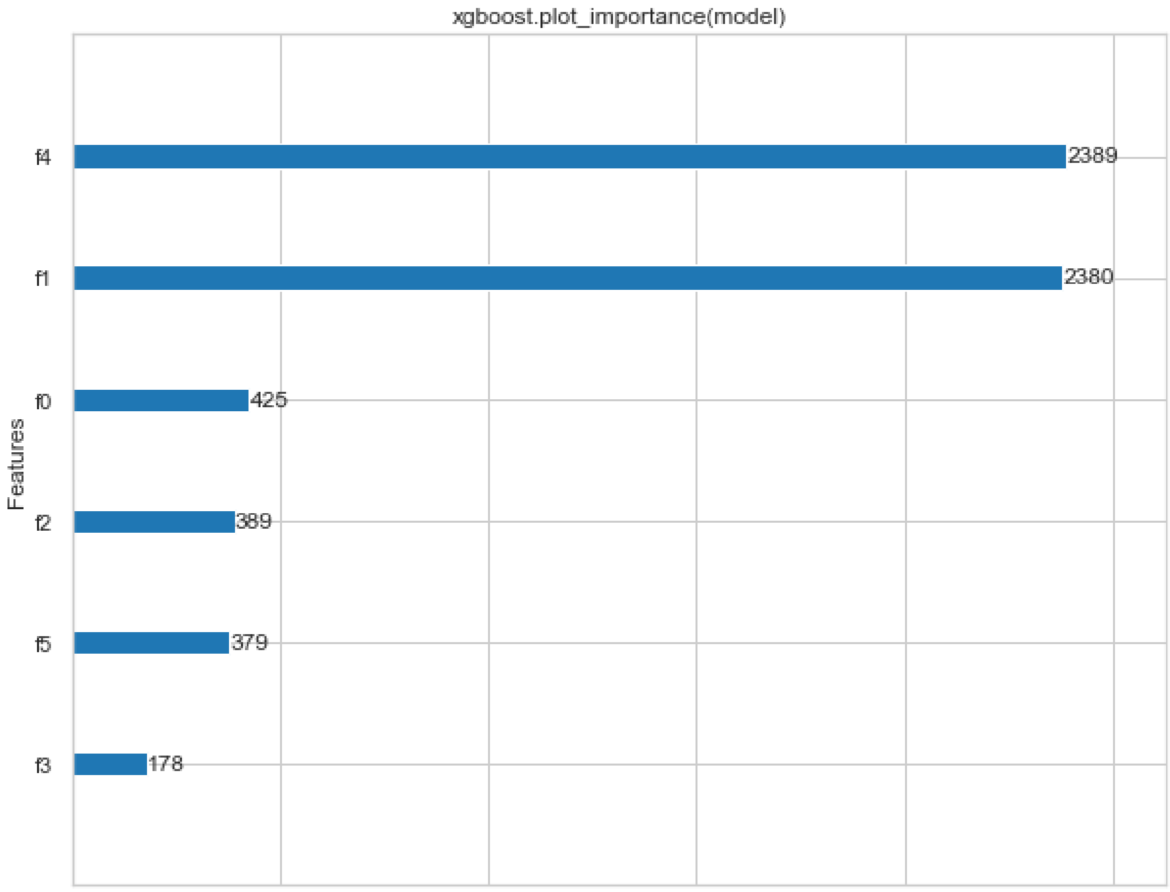
In [18]:

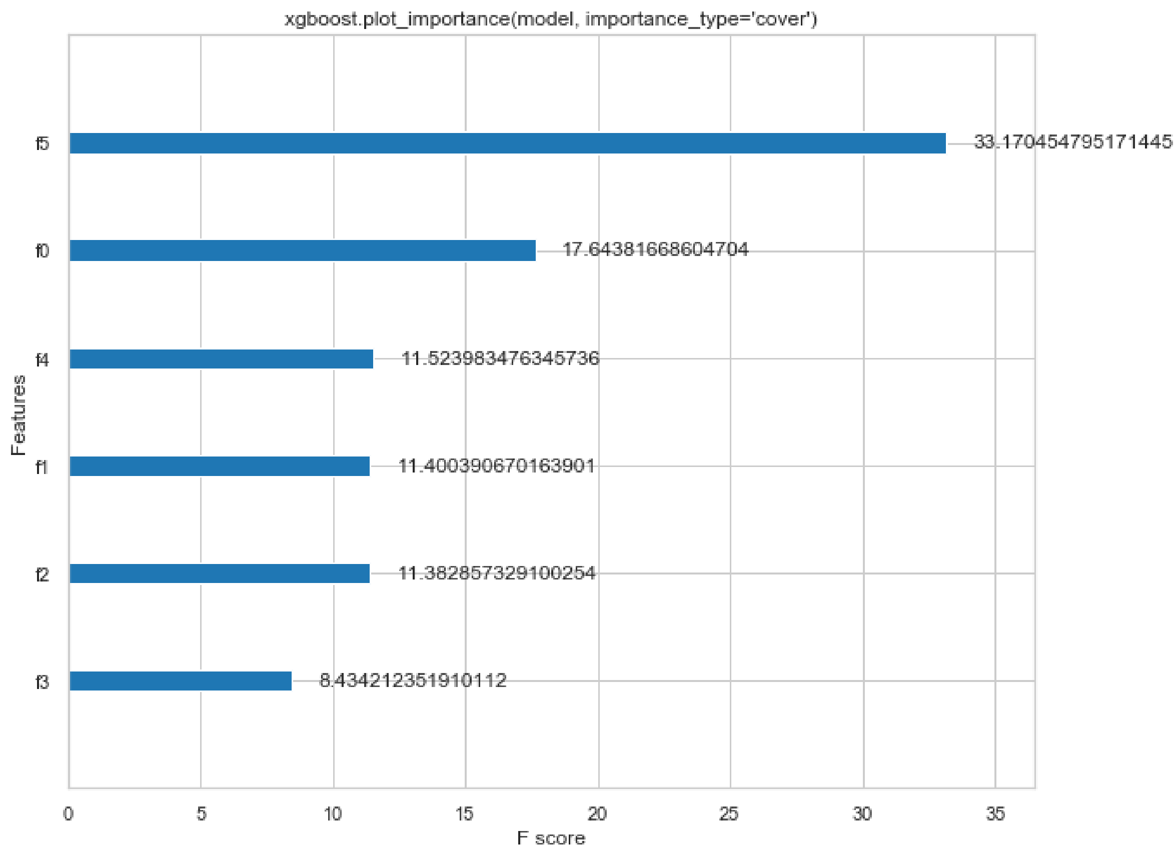
```
# b) Discuss Feature importance

def feat_imp(x,y):

    # Bar plot

    X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.4, random_state=7

    plt.figure(figsize = (15,9))

    seed = 7

    model = XGBClassifier()
    model.fit(X_train,y_train)

    for name, importance in zip(titanic.columns, model.feature_importances_):
        print(f'{name:15s}  {importance:.4f}')

    sns.barplot(x = titanic.columns[:-1], y=model.feature_importances_)


    # Sideways plot

    d_train = xgb.DMatrix(X_train, label=y_train)
    d_test = xgb.DMatrix(X_test, label=y_test)

    params = {
        "eta": 0.01,
        "objective": "binary:logistic",
        "subsample": 0.5,
        "base_score": np.mean(y_train),
        "eval_metric": "logloss"
    }

    model = xgb.train(params, d_train, 5000, evals = [(d_test, "test")], verbose_eval=100,

    ax = xgb.plot_importance(model, importance_type="weight")
    pl.title("xgboost.plot_importance(model)")

    print(f'Weight. The number of times a feature is used to split the data across all tree

    ax.figure.set_size_inches(10,8)

    ax = xgb.plot_importance(model, importance_type="cover")
    pl.title("xgboost.plot_importance(model, importance_type='cover')")

    ax.figure.set_size_inches(10,8)

    ax = xgb.plot_importance(model, importance_type="gain")
    pl.title("xgboost.plot_importance(model, importance_type='gain')")

    ax.figure.set_size_inches(10,8)

feat_imp(x,y)
```

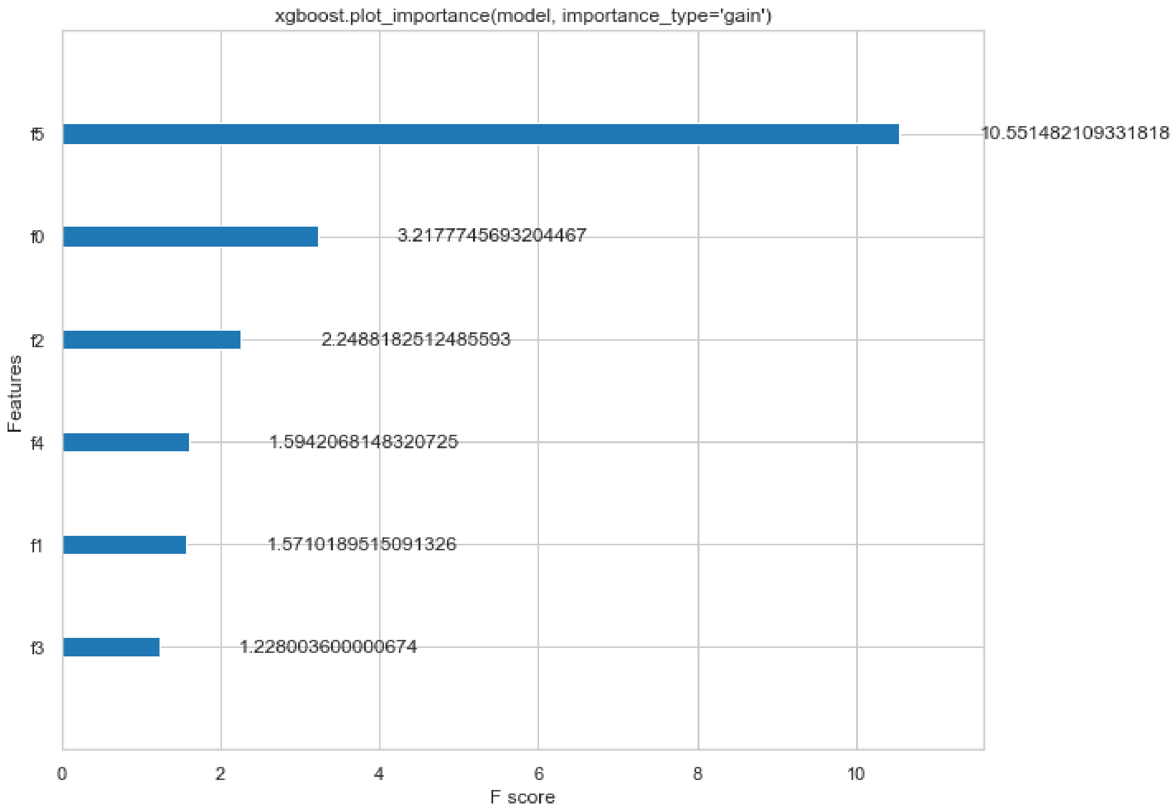
```
Survived         0.1863
Pclass           0.0639
```

```
Age                0.0874
Siblings/Spouses Aboard  0.0102
Parents/Children Aboard  0.0481
Fare               0.6041
[0]     test-logloss:0.675072
Will train until test-logloss hasn't improved in 20 rounds.
[100]   test-logloss:0.468551
[200]   test-logloss:0.419418
[300]   test-logloss:0.406387
Stopping. Best iteration:
[345]   test-logloss:0.403807

Weight. The number of times a feature is used to split the data across all
trees.
```




xgboost.plot_importance(model)
Features
f4
2389
f1
2380
f0
425
f2
389
f5
379
f3
178


xgboost.plot_importance(model, importance_type='cover')
Features
f5
33.170454795171445
f0
17.64381668604704
f4
11.523983476345736
f1
11.400390670163901
f2
11.382857329100254
f3
8.434212351910112
0
5
10
15
20
25
30
35
F score


xgboost.plot_importance(model, importance_type='gain')
f5
f0
f2
f4
f1
f3
Features
10.551482109331818
3.2177745693204467
2.2488182512485593
1.5942068148320725
1.5710189515091326
1.228003600000674
0
2
4
6
8
10
F score

In [ ]: