Dauphine UNIVERSITÉ PARIS | PSL

→ COMPUTER SCIENCE, DECISION-MAKING, AND DATA

Algorithmic and advanced Programming in Python

Remy Belmonte remy.belmonte@dauphine.eu

Lab 8

Using XGBoost in Python

XGBoost is one of the most popular machine learning algorithm these days. Regardless of the type of prediction task at hand; regression or classification.

Why XGBoost?

XGBoost is well known to provide better solutions than other machine learning algorithms. In fact, since its inception, it has become the "state-of-the-art” machine learning algorithm to deal with structured data.

In this tutorial, you’ll learn to build machine learning models using XGBoost in python. More specifically you will learn:

- what Boosting is and how XGBoost operates.
- how to apply XGBoost on a dataset and validate the results.
- about various hyper-parameters that can be tuned in XGBoost to improve model's performance.
- how to visualize the Boosted Trees and Feature Importance

But what makes XGBoost so popular?

- **Core algorithm is parallelizable** : Because the core XGBoost algorithm is parallelizable it can harness the power of multi- core computers. It is also parallelizable onto GPU's and across networks of computers making it feasible to train on very large datasets as well.

- **Consistently outperforms other algorithm methods** : It has shown better performance on a variety of machine learning benchmark datasets.

Wide variety of tuning parameters :
XGBoost internally has parameters for cross-validation, regularization, user-defined objective functions, missing values, tree parameters, scikit- learn compatible API etc.

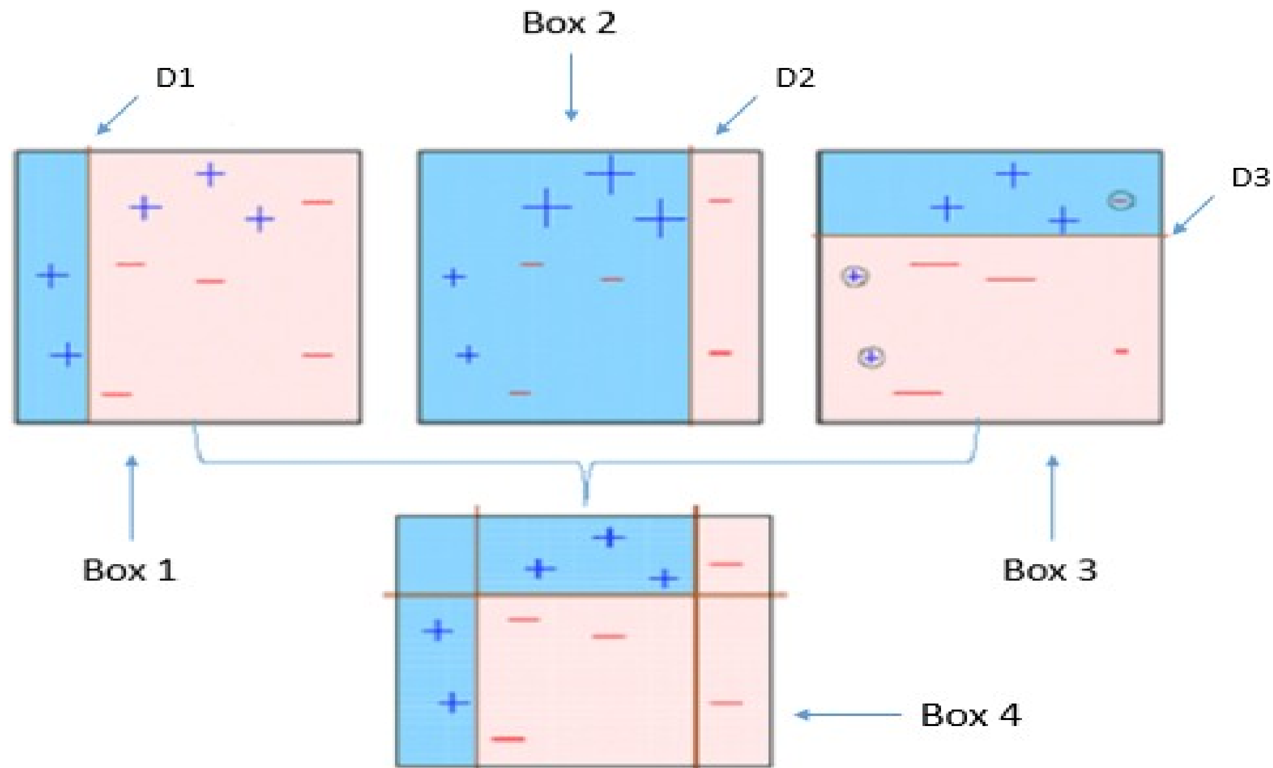
XGBoost (Extreme Gradient Boosting) belongs to a family of boosting algorithms and uses the gradient boosting (GBM) framework at its core. It is an optimized distributed gradient boosting library. But wait, what is boosting? Well, keep on reading.

Boosting

Boosting is a sequential technique which works on the principle of an ensemble. It combines a set of weak learners and delivers improved prediction accuracy. At any instant t, the model outcomes are weighed based on the outcomes of previous instant t-1. The outcomes predicted correctly are given a lower weight and the ones miss-classified are weighted higher.

Example

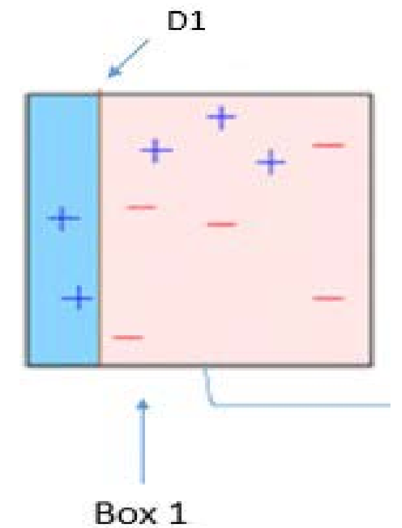
Let's understand boosting in general with a simple illustration.



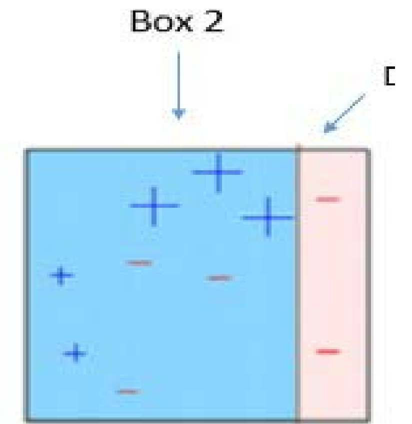
Four classifiers (in 4 boxes), shown above, are trying to classify + and - classes as homogeneously as possible.

1. **Box 1:** The first classifier (usually a decision stump) creates a vertical line (split) at D1. It says anything to the left of D1 is + and anything to the right of D1 is -. However, this classifier misclassifies three + points.

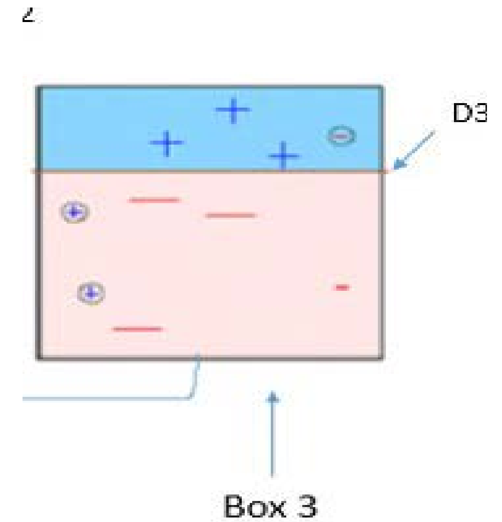
Note a Decision Stump is a Decision Tree model that only splits off at one level, therefore the final prediction is based on only one feature.



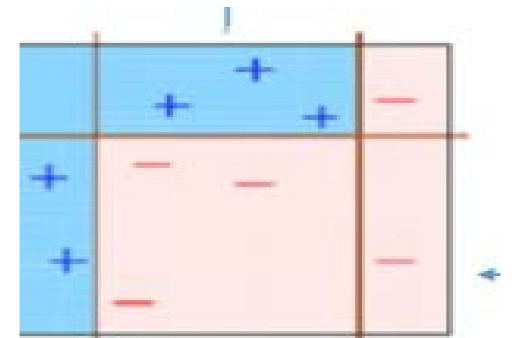
2. Box 2: The second classifier gives more weight to the three + misclassified points (see the bigger size of +) and creates a vertical line at D2. Again it says, anything to the right of D2 is - and left is +. Still, it makes mistakes by incorrectly classifying three - points.



3. Box 3: Again, the third classifier gives more weight to the three - misclassified points and creates a horizontal line at D3. Still, this classifier fails to classify the points (in the circles) correctly.



4. Box 4: This is a weighted combination of the weak classifiers (Box 1,2 and 3). As you can see, it does a good job at classifying all the points correctly.



That's the basic idea behind boosting algorithms is building a weak model, making conclusions about the various feature importance and parameters, and then using those conclusions to build a new, stronger model and capitalize on the misclassification error of the previous model and try to reduce it. Now, let's come to XGBoost.

Now, let's come to XGBoost.

To begin with, you should know about the default base learners of XGBoost: tree ensembles. The tree ensemble model is a set of classification and regression trees (CART). Trees are grown one after another, and attempts to reduce the misclassification rate are made in subsequent iterations.

Here's a simple example of a CART that classifies whether someone will like computer games straight from the XGBoost's documentation.

If you check the image in Tree Ensemble section, you will notice each tree gives a different prediction score depending on the data it sees and the scores of each individual tree are summed up to get the final score.

In this tutorial, you will be using XGBoost to solve a regression problem. The dataset is taken from the UCI Machine Learning Repository and is also present in sklearn's datasets module. It has 14 explanatory variables describing various aspects of residential homes in Boston, the challenge is to predict the median value of owner-occupied homes per $1000s.

Using XGBoost in Python

First of all, just like what you do with any other dataset, you are going to import the Boston Housing dataset and store it in a variable called boston. To import it from scikit-learn you will need to run this snippet.

```
from sklearn.datasets import load_boston
boston = load_boston()
```

The boston variable itself is a dictionary, so you can check for its keys using the .keys() method.

```
print(boston.keys())
> dict_keys(['data', 'target', 'feature_names', 'DESCR'])
```

You can easily check for its shape by using the boston.data.shape attribute, which will return the size of the dataset.

```
print(boston.data.shape)
> (506, 13)
```

As you can see it returned (506, 13), that means there are 506 rows of data with 13 columns. Now, if you want to know what the 13 columns are, you can simply use the .feature_names attribute and it will return the feature names.

```
print(boston.feature_names)
> ['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD' 'TAX'
'PTRATIO' 'B' 'LSTAT’]
```

The description of the dataset is available in the dataset itself. You can take a look at it using .DESCR .

```
print(boston.DESCR)
```

```
Boston House Prices dataset
===========================

Notes
------
Data Set Characteristics:

    :Number of Instances: 506

    :Number of Attributes: 13 numeric/categorical predictive

    :Median Value (attribute 14) is usually the target

    :Attribute Information (in order):
        - CRIM     per capita crime rate by town
        - ZN       proportion of residential land zoned for lots over 25,000 sq.ft.
        - INDUS    proportion of non-retail business acres per town
        - CHAS     Charles River dummy variable (= 1 if tract bounds river; 0 otherwise)
        - NOX      nitric oxides concentration (parts per 10 million)
```

Now let's convert it into a pandas DataFrame! For that you need to import the `pandas` library and call the `DataFrame()` function passing the argument `boston.data`. To label the names of the columns, use the .columnns attribute of the pandas DataFrame and assign it to boston.feature_names

```python
import pandas as pd

data = pd.DataFrame(boston.data)

data.columns = boston.feature_names
```

Explore the top 5 rows of the dataset by using head() method on your pandas DataFrame.

```python
data.head()
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33

You'll notice that there is no column called PRICE in the DataFrame. This is because the target column is available in another attribute called boston. target . Append boston. target to your pandas DataFrame.

```
data['PRICE'] = boston.target
```

Run the .info() method on your DataFrame to get useful information about the data.

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 506 entries, 0 to 505
Data columns (total 14 columns):
CRIM         506 non-null float64
ZN           506 non-null float64
INDUS        506 non-null float64
CHAS         506 non-null float64
NOX          506 non-null float64
```

```
RM          506 non-null float64
AGE         506 non-null float64
DIS         506 non-null float64
RAD         506 non-null float64
TAX         506 non-null float64
PTRATIO     506 non-null float64
B           506 non-null float64
LSTAT       506 non-null float64
PRICE       506 non-null float64
dtypes: float64(14)
memory usage: 55.4 KB
```

Turns out that this dataset has 14 columns (including the target variable `PRICE`) and 506 rows. Notice that the columns are of `float` data-type indicating the presence of only continuous features with no missing values in any of the columns. To get more summary statistics of the different features in the dataset you will use the `describe()` method on your DataFrame.

Note that `describe()` only gives summary statistics of columns which are continuous in nature and not categorical.

```
data.describe()
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	PRICE
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000
mean	3.593761	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901	3.795043	9.549407	408.237154	18.455534	356.674032	12.653063	22.532806
std	8.596783	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861	2.105710	8.707259	168.537116	2.164946	91.294864	7.141062	9.197104
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1.129600	1.000000	187.000000	12.600000	0.320000	1.730000	5.000000
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000	2.100175	4.000000	279.000000	17.400000	375.377500	6.950000	17.025000
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000	3.207450	5.000000	330.000000	19.050000	391.440000	11.360000	21.200000
75%	3.647423	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000	5.188425	24.000000	666.000000	20.200000	396.225000	16.955000	25.000000
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12.126500	24.000000	711.000000	22.000000	396.900000	37.970000	50.000000

If you plan to use XGBoost on a dataset which has categorical features you may want to consider applying some encoding (like one-hot encoding) to such features before training the model. Also, if you have some missing values such as NA in the dataset you may or may not do a separate treatment for them, because XGBoost is capable of handling missing values internally. You can check out this link if you wish to know more on this.

Without delving into more exploratory analysis and feature engineering, you will now focus on applying the algorithm to train the model on this data.

You will build the model using Trees as base learners (which are the default base learners) using XGBoost's scikit-learn compatible API. Along the way, you will also learn some of the common tuning parameters which XGBoost provides in order to improve the model's performance, and using the root mean squared error (RMSE) performance metric to check the performance of the trained model on the test set. Root mean Squared error is the square root of the mean of the squared

differences between the actual and the predicted values. As usual, you start by importing the library `xgboost` and other important libraries that you will be using for building the model.

Note you can install python libraries like `xgboost` on your system using `pip install xgboost` on cmd.

```
import xgboost as xgb
from sklearn.metrics import mean_squared_error
import pandas as pd
import numpy as np
```

Separate the target variable and rest of the variables using `.iloc` to subset the data.

```
X, y = data.iloc[:,:-1],data.iloc[:,-1]
```

Now you will convert the dataset into an optimized data structure called `Dmatrix` that XGBoost supports and gives it acclaimed performance and efficiency gains. You will use this later in the tutorial.

```
data_dmatrix = xgb.DMatrix(data=X,label=y)
```

XGBoost's hyperparameters

At this point, before building the model, you should be aware of the tuning parameters that XGBoost provides. Well, there are a plethora of tuning parameters for tree-based learners in XGBoost and you can read all about them here. But the most common ones that you should know are:

- learning_rate : step size shrinkage used to prevent overfitting. Range is [0,1]
- max_depth : determines how deeply each tree is allowed to grow during any boosting round.

- subsample : percentage of samples used per tree. Low value can lead to underfitting.

- colsample_bytree : percentage of features used per tree. High value can lead to overfitting.

- n_estimators : number of trees you want to build.

- objective : determines the loss function to be used like reg:linear for regression problems, reg:logistic for classification problems with only decision, binary:logistic for classification problems with probability.

XGBoost also supports regularization parameters to penalize models as they become more complex and reduce them to simple (parsimonious) models.

- gamma : controls whether a given node will split based on the expected reduction in loss after the split. A higher value leads to fewer splits. Supported only for tree-based learners.

- alpha : L1 regularization on leaf weights. A large value leads to more regularization.

- lambda : L2 regularization on leaf weights and is smoother than L1 regularization.

It's also worth mentioning that though you are using trees as your base learners, you can also use XGBoost's relatively less popular linear base learners and one other tree learner known as dart. All you have to do is set the booster parameter to either gbtree (default), gblinear or dart.

Now, you will create the train and test set for cross-validation of the results using the train_test_split function from sklearn's model_selection module with test_size size equal to 20% of the data. Also, to maintain reproducibility of the results, a random_state is also assigned.

```
from sklearn.model_selection import train_test_split
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_
```

The next step is to instantiate an XGBoost regressor object by calling the `XGBRegressor()` class from the XGBoost library with the hyper-parameters passed as arguments. For classification problems, you would have used the `XGBClassifier()` class.

```
xg_reg = xgb.XGBRegressor(objective ='reg:linear', colsample_by
                max_depth = 5, alpha = 10, n_estimators = 10)
```

Fit the regressor to the training set and make predictions on the test set using the familiar `.fit()` and `.predict()` methods.

```
xg_reg.fit(X_train,y_train)

preds = xg_reg.predict(X_test)
```

Compute the rmse by invoking the `mean_sqaured_error` function from sklearn's `metrics` module.

```
rmse = np.sqrt(mean_squared_error(y_test, preds))
print("RMSE: %f" % (rmse))
```

```
RMSE: 10.569356
```

Well, you can see that your RMSE for the price prediction came out to be around 10.8 per 1000$.

k-fold Cross Validation using XGBoost

```
rmse = np.sqrt(mean_squared_error(y_test, preds))
print("RMSE: %f" % (rmse))
```

```
RMSE: 10.569356
```

Well, you can see that your RMSE for the price prediction came out to be around 10.8 per 1000$.

k-fold Cross Validation using XGBoost

In order to build more robust models, it is common to do a k-fold cross validation where all the entries in the original

training dataset are used for both training as well as validation. Also, each entry is used for validation just once. XGBoost supports k-fold cross validation via the `cv()` method. All you have to do is specify the `nfolds` parameter, which is the number of cross validation sets you want to build.

k-fold Cross Validation using XGBoost

Also, it supports many other parameters (check out this link) like:

- num_boost_round : denotes the number of trees you build (analogous to n_estimators)
- metrics : tells the evaluation metrics to be watched during CV
- as_pandas : to return the results in a pandas DataFrame.

- `early_stopping_rounds` : finishes training of the model early if the hold-out metric ("rmse" in our case) does not improve for a given number of rounds.

- `seed` : for reproducibility of results.

This time you will create a hyper-parameter dictionary `params` which holds all the hyper-parameters and their values as key-value pairs but will exclude the `n_estimators` from the hyper-

parameter dictionary because you will use num_boost_rounds instead.

You will use these parameters to build a 3-fold cross validation model by invoking XGBoost's cv() method and store the results in a cv_results DataFrame. Note that here you are using the Dmatrix object you created before.

```
params = {"objective":"reg:linear",'colsample_bytree': 0.3,'lea
                'max_depth': 5, 'alpha': 10}

cv_results = xgb.cv(dtrain=data_dmatrix, params=params, nfold=3
                    num_boost_round=50,early_stopping_rounds=10
```

`cv_results` contains train and test RMSE metrics for each boosting round.

```
cv_results.head()
```

	test-rmse-mean	test-rmse-std	train-rmse-mean	train-rmse-std
0	21.746693	0.019311	21.749371	0.033853
1	19.891096	0.053295	19.859423	0.029633
2	18.168509	0.014465	18.072169	0.018803
3	16.687861	0.037342	16.570206	0.018556
4	15.365013	0.059400	15.206344	0.015451

Extract and print the final boosting round metric.

```
print((cv_results["test-rmse-mean"]).tail(1))
```

```
49    4.031162
Name: test-rmse-mean, dtype: float64
```

You can see that your RMSE for the price prediction has reduced as compared to last time and came out to be around 4.03 per 1000$. You can reach an even lower RMSE for a different set of hyper-parameters. You may consider applying techniques like Grid Search, Random Search and Bayesian Optimization to reach the optimal set of hyper-parameters.

Visualize Boosting Trees and Feature Importance

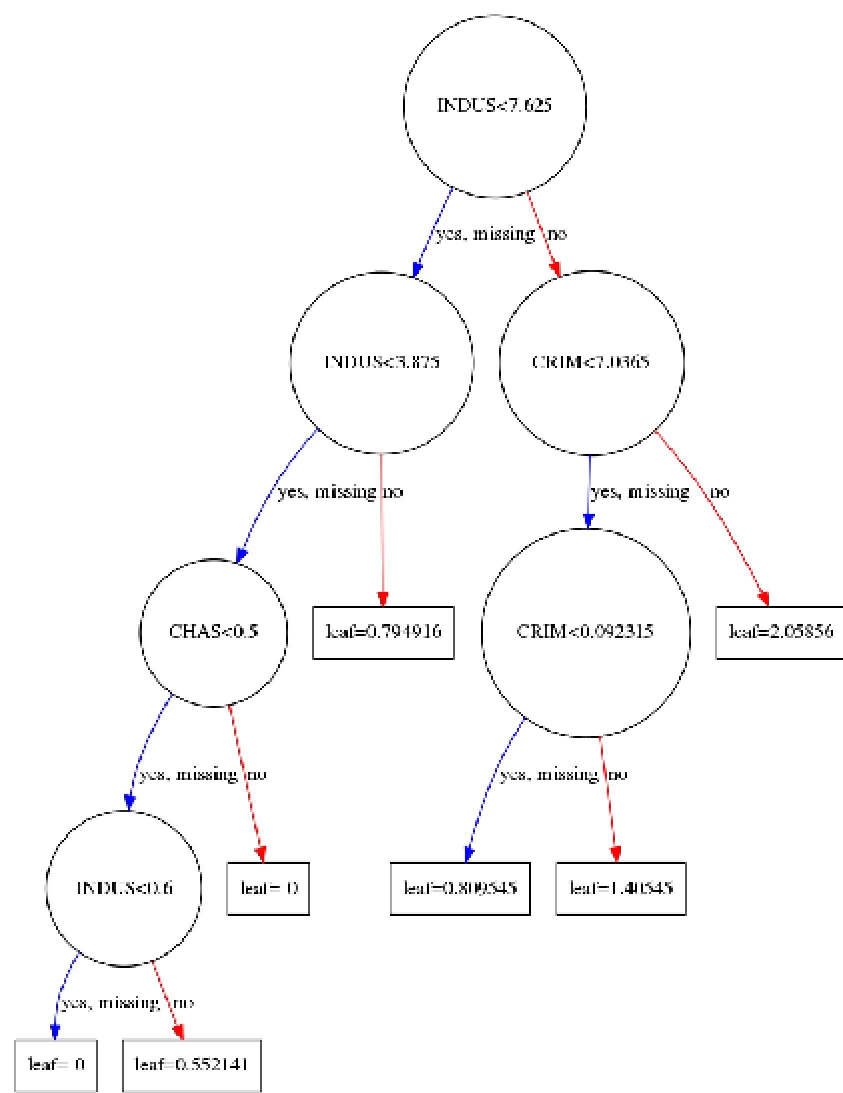
You can also visualize individual trees from the fully boosted model that XGBoost creates using the entire housing dataset. XGBoost has a plot_tree() function that makes this type of visualization easy. Once you train a model using the XGBoost learning API, you can pass it to the plot_tree() function along with the number of trees you want to plot using the num_trees argument.

```
xg_reg = xgb.train(params=params,
dtrain=data_dmatrix, num_boost_round=10)
```

Plotting the first tree with the `matplotlib` library:

```
import matplotlib.pyplot as plt

xgb.plot_tree(xg_reg,num_trees=0)
plt.rcParams['figure.figsize'] = [50, 10]
plt.show()
```


INDUS<7.625
yes, missing
no
INDUS<3.875
CRIM<7.0365
yes, missing
no
yes, missing
no
CHAS<0.5
leaf=0.794916
CRIM<0.092315
leaf=2.05856
yes, missing
no
yes, missing
no
INDUS<0.6
leaf= 0
yes, missing
no
leaf= 0
leaf=0.552141

These plots provide insight into how the model arrived at its final decisions and what splits it made to arrive at those decisions.

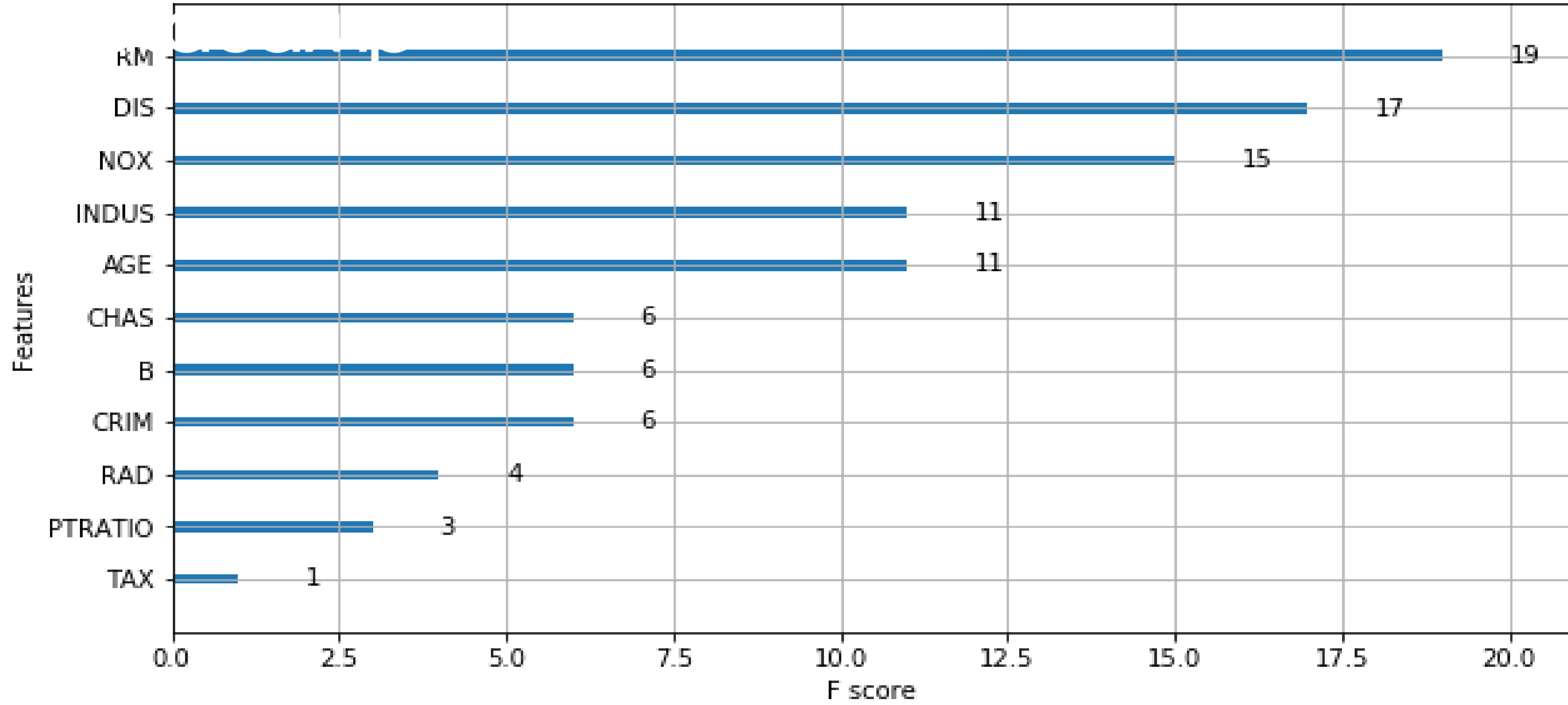
Note that if the above plot throws the 'graphviz' error on your system, consider installing the graphviz package via `pip install graphviz` on cmd. You may also need to run `sudo apt-get install graphviz` on cmd. (link)

Another way to visualize your XGBoost models is to examine the importance of each feature column in the original dataset within the model.

One simple way of doing this involves counting the number of times each feature is split on across all boosting rounds (trees) in the model, and then visualizing the result as a bar graph, with the features ordered according to how many times they appear. XGBoost has a `plot_importance()` function that allows

you to do exactly this.

```
xgb.plot_importance(xg_reg)
plt.rcParams['figure.figsize'] = [5, 5]
plt.show()
```


Feature importance
RM
DIS
NOX
INDUS
AGE
CHAS
B
CRIM
RAD
PTRATIO
TAX
19
17
15
11
11
6
6
6
4
3
1
Features
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0
F score

As you can see the feature RM has been given the highest importance score among all the features. Thus XGBoost also gives you a way to do Feature Selection. Isn't this brilliant?

Conclusion

In this lag you have learned how to use XGBoost
You started off with understanding how Boosting works in general and then narrowed down to XGBoost specifically. You also practiced applying XGBoost on an open source dataset and along the way you learned about its hyper-parameters, doing cross-validation, visualizing the trees and in the end how it can also be used as a Feature Selection technique

In Lab session

You will see how to use XGBoost to do price prediction for houses in Boston
This can be useful for your **FINAL** project

Lab is done by Remy Belmonte