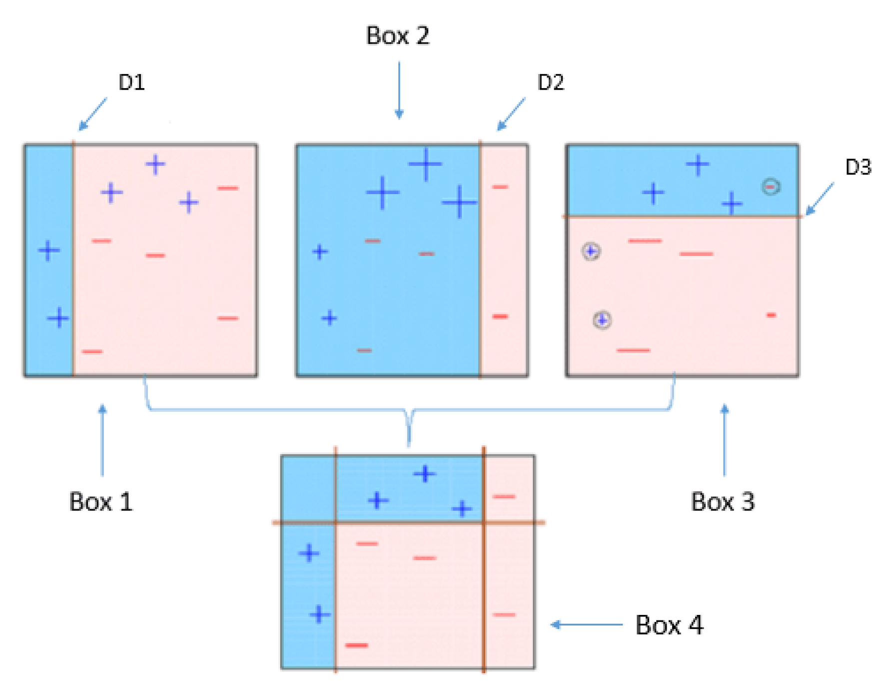
XGBoost Exercise with Boston Dataset

The goal of this exercise is to practice with the XGBoost algorithm.

How does **XGBoost** work? Here are some details explaining how it works:

It is based on **Boosting**, being a sequential technique working as 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.

Here is an illustration of the Boosting Model:



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.

1. **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.
2. **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.
3. **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.

XGBoost

The default base learners of XGBoost are **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. 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 [1]:
```
# Loading the dataset
from sklearn.datasets import load_boston
boston = load_boston()
```

In [2]:
```
# Keys of the dictionary
print(boston.keys())
```

```
dict_keys(['data', 'target', 'feature_names', 'DESCR', 'filename'])
```

In [3]:
```
print(boston.data.shape)
```

```
(506, 13)
```

In [4]:
```
print(boston.feature_names)
```

```
['CRIM' 'ZN' 'INDUS' 'CHAS' 'NOX' 'RM' 'AGE' 'DIS' 'RAD' 'TAX' 'PTRATIO'
 'B' 'LSTAT']
```

In [5]:
```
print(boston.DESCR)
```

```
.. _boston_dataset:

Boston house prices dataset
---------------------------

**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)
        - RM       average number of rooms per dwelling
        - AGE      proportion of owner-occupied units built prior to 1940
        - DIS      weighted distances to five Boston employment centres
        - RAD      index of accessibility to radial highways
        - TAX      full-value property-tax rate per $10,000
        - PTRATIO  pupil-teacher ratio by town
        - B        1000(Bk - 0.63)^2 where Bk is the proportion of blacks by town
        - LSTAT    % lower status of the population
        - MEDV     Median value of owner-occupied homes in $1000's

    :Missing Attribute Values: None

    :Creator: Harrison, D. and Rubinfeld, D.L.

This is a copy of UCI ML housing dataset.
https://archive.ics.uci.edu/ml/machine-learning-databases/housing/


This dataset was taken from the StatLib library which is maintained at Carnegie Mellon University.

The Boston house-price data of Harrison, D. and Rubinfeld, D.L. 'Hedonic
prices and the demand for clean air', J. Environ. Economics & Management,
vol.5, 81-102, 1978.   Used in Belsley, Kuh & Welsch, 'Regression diagnostics
...', Wiley, 1980.   N.B. Various transformations are used in the table on
pages 244-261 of the latter.

The Boston house-price data has been used in many machine learning papers that address regression
problems.

.. topic:: References

   - Belsley, Kuh & Welsch, 'Regression diagnostics: Identifying Influential Data and Sources of Coll
inearity', Wiley, 1980. 244-261.
   - Quinlan,R. (1993). Combining Instance-Based and Model-Based Learning. In Proceedings on the Tent
h International Conference of Machine Learning, 236-243, University of Massachusetts, Amherst. Morgan
Kaufmann.
```

In [6]:
```
import pandas as pd
# Converting the data into Pandas DataFrame
data = pd.DataFrame(boston.data)
data.columns = boston.feature_names
```

In [7]:
```
data.head()
```

Out[7]:

	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

Obtaining the **PRICE** variable from the 'target' attribute:

In [8]:
```
data['PRICE'] = boston.target
```

In [9]:
```
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: 55.5 KB
```

In [10]:
```
data.describe()
```

Out[10]:

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRAT
count	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.000000	506.0000
mean	3.613524	11.363636	11.136779	0.069170	0.554695	6.284634	68.574901	3.795043	9.549407	408.237154	18.4555
std	8.601545	23.322453	6.860353	0.253994	0.115878	0.702617	28.148861	2.105710	8.707259	168.537116	2.1649
min	0.006320	0.000000	0.460000	0.000000	0.385000	3.561000	2.900000	1.129600	1.000000	187.000000	12.6000
25%	0.082045	0.000000	5.190000	0.000000	0.449000	5.885500	45.025000	2.100175	4.000000	279.000000	17.4000
50%	0.256510	0.000000	9.690000	0.000000	0.538000	6.208500	77.500000	3.207450	5.000000	330.000000	19.0500
75%	3.677083	12.500000	18.100000	0.000000	0.624000	6.623500	94.075000	5.188425	24.000000	666.000000	20.2000
max	88.976200	100.000000	27.740000	1.000000	0.871000	8.780000	100.000000	12.126500	24.000000	711.000000	22.0000

In [11]:
```
# Importing some useful libraries
import xgboost as xgb
from sklearn.metrics import mean_squared_error
import pandas as pd
import numpy as np
```

In [12]:
```
# Separating the target variable from the other variables
X, y = data.iloc[:,:-1],data.iloc[:,-1]
```

In [13]:
```
# Converting the dataset into an optimized structure supported by XGBoost
data_dmatrix = xgb.DMatrix(data=X,label=y)
```

XGBoost Hyperparameters

There are some popular hyperparameters:

- **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**.

In [14]:
```
# Splitting the data
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=123)
```

In [15]:
```
# Instanctiating an XGBoost regressor object
xg_reg = xgb.XGBRegressor(objective ='reg:linear', colsample_bytree = 0.3, learning_rate = 0.1,
                max_depth = 5, alpha = 10, n_estimators = 10)
```

In [16]:
```
# Fitting the regressor to the training set
xg_reg.fit(X_train,y_train)
# Making the predictions
preds = xg_reg.predict(X_test)
```

```
[13:05:22] WARNING: C:/Users/Administrator/workspace/xgboost-win64_release_1.5.1/src/objective/regres
sion_obj.cu:188: reg:linear is now deprecated in favor of reg:squarederror.
```

In [17]:
```
rmse = np.sqrt(mean_squared_error(y_test, preds))
print("RMSE: %f" % (rmse))
```

```
RMSE: 10.423243
```

k-fold Cross Validation

In this process, 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 that has to be done is specify the **nfolds** parameter, which is the number of cross validation sets you want to build. Also, it supports many other parameters 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 it is displayed the creation of 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 num_boost_rounds will be used instead.

These parameters will be used to build a 3-fold cross validation model by invoking XGBoost's cv() method and store the results in a cv_results DataFrame.

In [18]:
```
params = {"objective":"reg:linear",'colsample_bytree': 0.3,'learning_rate': 0.1,
                'max_depth': 5, 'alpha': 10}

cv_results = xgb.cv(dtrain=data_dmatrix, params=params, nfold=3,
                    num_boost_round=50,early_stopping_rounds=10,metrics="rmse", as_pandas=True, seed=12
3)
```

```
[13:05:22] WARNING: C:/Users/Administrator/workspace/xgboost-win64_release_1.5.1/src/objective/regres
sion_obj.cu:188: reg:linear is now deprecated in favor of reg:squarederror.
[13:05:22] WARNING: C:/Users/Administrator/workspace/xgboost-win64_release_1.5.1/src/objective/regres
sion_obj.cu:188: reg:linear is now deprecated in favor of reg:squarederror.
[13:05:22] WARNING: C:/Users/Administrator/workspace/xgboost-win64_release_1.5.1/src/objective/regres
sion_obj.cu:188: reg:linear is now deprecated in favor of reg:squarederror.
```

In [19]:
```
cv_results.head()
```

Out[19]:

	train-rmse-mean	train-rmse-std	test-rmse-mean	test-rmse-std
0	21.750757	0.036152	21.765523	0.028850
1	19.778531	0.077649	19.830760	0.031761
2	18.052811	0.118631	18.157337	0.116038
3	16.458958	0.169188	16.623975	0.191413
4	15.074781	0.183545	15.254608	0.213612

In [20]:
```
print((cv_results["test-rmse-mean"]).tail(1))
```

```
49    3.99692
Name: test-rmse-mean, dtype: float64
```

In [21]:
```
xg_reg = xgb.train(params=params, dtrain=data_dmatrix, num_boost_round=10)
```

```
[13:05:22] WARNING: C:/Users/Administrator/workspace/xgboost-win64_release_1.5.1/src/objective/regres
sion_obj.cu:188: reg:linear is now deprecated in favor of reg:squarederror.
```

In [22]:
```
import matplotlib.pyplot as plt
import os
os.environ["PATH"] += os.pathsep + 'C:/Users/wavm0/OneDrive/Documents/Graphviz/bin/'

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

```
<Figure size 640x480 with 1 Axes>
```

In [23]:
```
xgb.plot_importance(xg_reg)
plt.rcParams['figure.figsize'] = [5, 5]
plt.show()
```