Table of Contents

1
6. Classification and Regression with Trees

Tree based algorithms are very popular due to two reasons - they are interpretable & they make sound predictions that has won many machine learning competitions on online platforms such as Kaggle. Furthermore, they have many use-cases outside machine learning in order to solve problems that are both simple & complex.

Building a "tree" is an approach to decision making used in almost all industries. They can be used to solve both classification & regression based problems & has several use-cases that make it the go-to solution!

This chapter is broadly divided into two sections:

1. Classification Trees
2. Regression Trees

Each section will cover the fundamental theory of the different types of tree based algorithms along with their implementation in scikit-learn. In the end of this chapter you will learn how to aggregate several algorithms into an "ensemble" and have them vote on what the best prediction is.

Classification Trees

Classification trees are the trees that are used to predict a category or class. This is similar to the classification algorithms that you have previously learned about in this book such as the K-Nearest Neighbors or the Logistic Regression.

Broadly speaking there are three tree based algorithms that are used to solve classification problems:

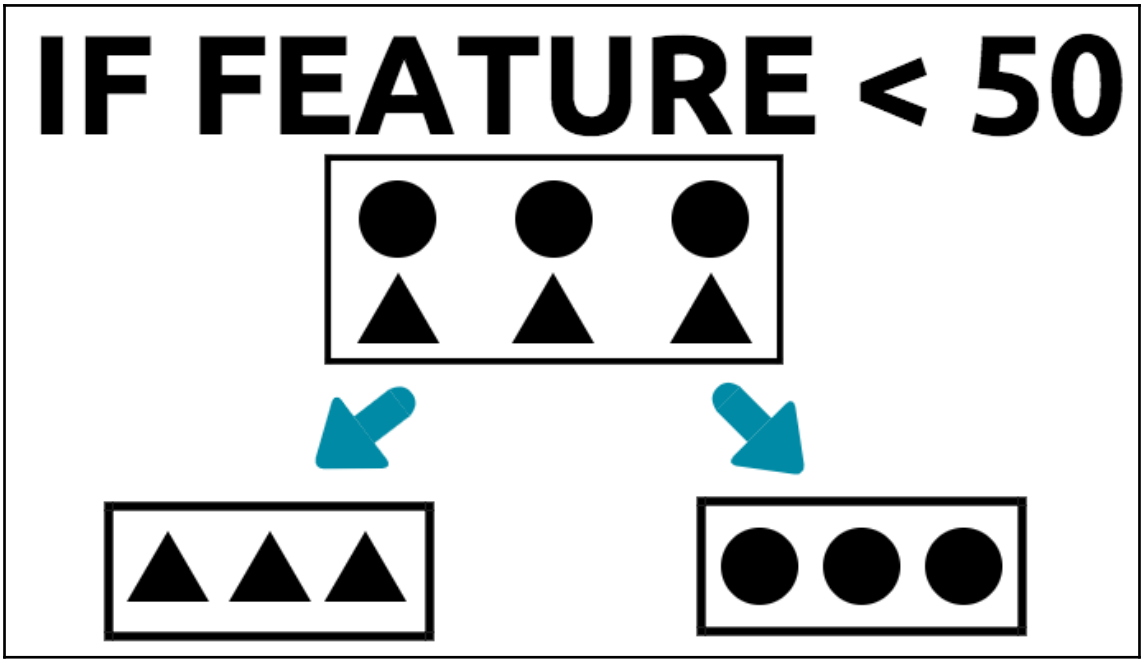
1. The Decision Tree Classifier

2. The Random Forest Classifier
3. The AdaBoost Classifier

In this section you will learn about how each one of these tree based algorithm works in order to classify a row of data as a particular class/category.

The Decision Tree Classifier

The Decision Tree is the most simple tree based algorithm and serves as the foundation for the other three algorithms. Let's consider the simple decision tree shown below:



Simple decision tree

A decision tree, in simple terms is a set of rules that help us classify observations into distinct groups. In the image illustrated above the rule would be:

If (Value of Feature is lesser than 50) then (Put the triangles in the left box & Put the circles in the right box).

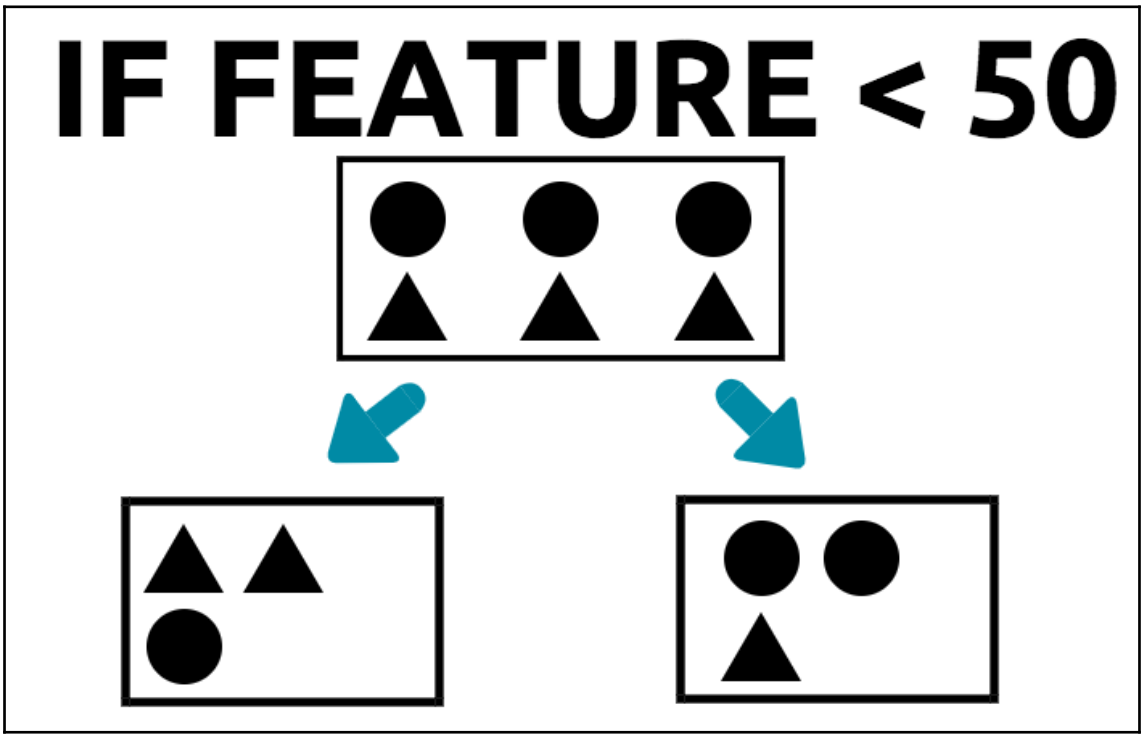
The decision tree above perfectly divides the observations into two distinct groups which is

the characteristic of an ideal decision tree. The first box on the top is called the 'root' of the tree and according to the decision tree this is the most important feature that is used to decide which groups the observations in that box goes into.

The boxes under the root node are known as the 'children'. In the tree above, the 'children' are also the 'leaf' node. The 'leaf' is the last set of boxes that will find in the decision tree and is usually in the bottom most part of the tree. As you might have guessed, the decision tree represents a regular tree but in the 'inverted' format.

Picking the best feature

How does the decision tree decide which feature is the best? The best feature is one that offers the best possible split and divides the tree into two or more distinct groups depending on the number of classes/categories that we have in the data. Let's have a look at the image shown below:

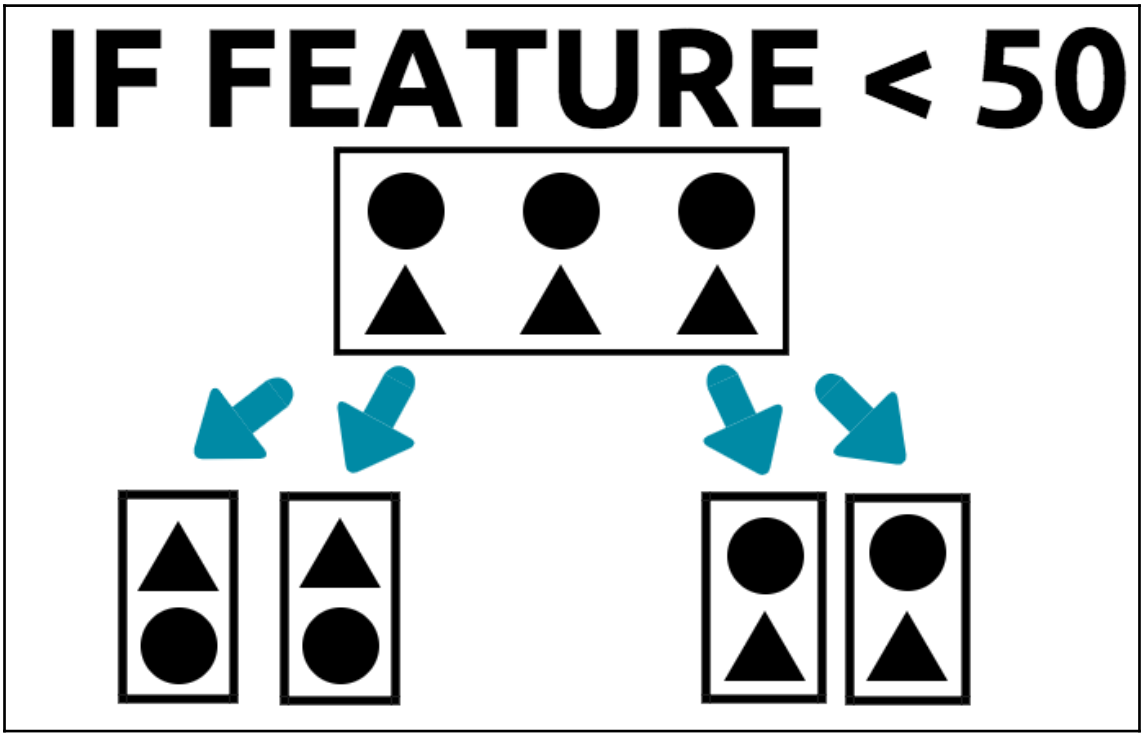


A good split

In the image above:

1. The tree splits the data in the 'root' node into two distinct groups.
2. In the left side group, we see that there are two triangles and one circle.
3. In the right side group, we that there are two circles and one triangle.
4. Since the tree got the majority of one class into one group (2/3 triangles and 2/3 circles on the right) we say that the tree has done a good job when it comes to splitting the data into distinct groups.

Let's take a look at another example in which the split is bad. This is illustrated in the image shown below:



A bad split

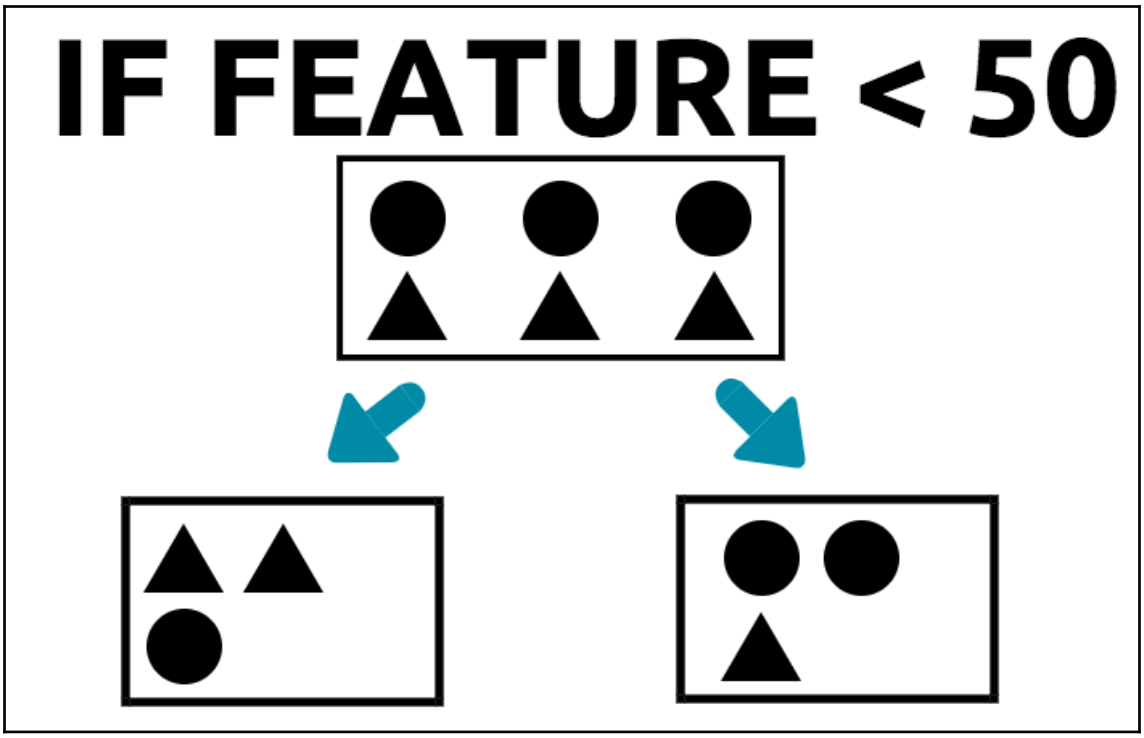
In the image above:

1. The tree splits the data in the 'root' node into 4 distinct groups. This is bad in itself as it is clear that there are only two categories (The circle & the triangle)
2. Furthermore, each group has 1 triangle and 1 circle
3. There is no majority class/category in any one of the 4 groups. Each group has

50% of one category, therefore the tree cannot come to a conclusive decision unless it relies on more features which then increases the complexity of the tree.

The 'Gini' coefficient

The metric that the decision tree uses to decide if the 'root' node is called the 'Gini' coefficient. A higher value of the this coefficient tells the tree that the particular feature does a fantastic job at splitting the data into distinct groups. In order to compute the 'Gini' coefficient for a feature let's consider the image shown below:



Computing the Gini coefficient

In the image above:

1. The feature splits the data into two groups.
2. In the left group we have two triangles and one circle.
3. Therefore, the Gini for the left group is = (2 triangles/3 total data points)^2+ (1 circle/3 total data points)^2.

4. This is = $2/3^2 + 1/3^2 = 0.55$
5. A value of 0.55 for the Gini coefficient indicates that the root of this tree splits the data in such a way that each group has a majority category.
6. A perfect root feature would have a Gini coefficient of 1. This means that each group has only one class/category.
7. A bad root feature would have a Gini coefficient of 0.5 which indicates that there is no distinct class/category in a group.

In reality, the decision tree is built in a recursive manner with the tree picking a random attribute for the root and then computing the Gini coefficient for that attribute. It does this until it finds the attribute that best splits the data in a node into groups that have distinct classes/categories.

Implementing the decision tree classifier in scikit-learn

In this section you will learn how to implement the decision tree classifier in scikit-learn. We will work with the same fraud detection dataset. The first step is to load in the dataset into the Jupyter Notebook. We can do this by using the code shown below:

```
import pandas as pd

df = pd.read_csv('fraud_prediction.csv')
```

The next step is to split the data into training and test sets. We can do that using the code shown below:

```
#Creating the features

features = df.drop('isFraud', axis = 1).values
target = df['isFraud'].values

X_train, X_test, y_train, y_test = train_test_split(features, target,
test_size = 0.3, random_state = 42, stratify = target)
```

We can now build the initial decision tree classifier on the training data and test it's accuracy on the test data by using the code shown below:

```
from sklearn.tree import DecisionTreeClassifier

dt = DecisionTreeClassifier(criterion = 'gini', random_state = 50)

#Fitting on the training data

dt.fit(X_train, y_train)
```

```
#Testing accuracy on the test data

dt.score(X_test, y_test)
```

In the code above:

1. We first import the *DecisionTreeClassifier* from scikit-learn
2. We then initialize a *DecisionTreeClassifier* object with two arguments - the 'criterion' is the metric that the tree uses to pick the most important features in a recursive manner which in this case is the Gini coefficient and the 'random_state' which is set to 50 so that the model produces the same result every time we run it.
3. Finally, we fit the model on the training data and evaluate it's accuracy on the test data.

Hyper-parameter tuning for the decision tree

The decision tree has a plethora of hyper-parameters that require fine-tuning in order to derive the best possible model that reduces the generalization error as much as possible. In this section we will focus on two specific hyper-parameters:

1. **Max Depth**: This is the maximum number of children nodes that can grow out from the decision tree until the tree is cut off. For example, if this is set to 3, then the tree will use 3 children nodes and cut the tree off before it can grow anymore.
2. **Min Samples Leaf:** This is the minimum number of samples/data points that are required to be present in the leaf node. The leaf node is the last node of the tree. This parameter, if set to a value of 0.04 tells the tree that the tree must grow until the last node contains 4% of the total samples in the data.

In order to optimize the ideal hyper-parameter and to extract the best possible decision tree we use the GridSearchCV module from scikit-learn by using the code shown below:

```
from sklearn.model_selection import GridSearchCV

#Creating a grid of different hyper-parameters

grid_params = {
    'max_depth': [1,2,3,4,5,6],
    'min_samples_leaf': [0.02,0.04, 0.06, 0.08]
}

#Building a 10 fold Cross Validated GridSearchCV object
```

```
grid_object = GridSearchCV(estimator = dt, param_grid = grid_params,
scoring = 'accuracy', cv = 10, n_jobs = -1)
```

In the code above:

1. We first import the *GridSearchCV* module from scikit-learn.
2. Next, we create a dictionary of possible values for the hyper-parameters and store it as "grid_params".
3. Finally, we create a *GridSearchCV* object with with the decision tree classifier as the estimator, the dictionary of hyper-parameter values.
4. We set the "scoring" argument as accuracy since we want to extract the accuracy of the best model according to GridSearchCV.

We then fit this grid object to the training data by using the code shown below:

```
#Fitting the grid to the training data

grid_object.fit(X_train, y_train)
```

We can then extract the best set of parameters by using the code shown below:

```
#Extracting the best parameters

grid_object.best_params_
```

The code above has indicated that a maximum depth of 1 and a minimum samples at the leaf node of 0.02 are the best parameters for this data. We can use these optimal parameters and construct a new decision tree by using the code shown below:

```
#Extracting the best parameters

grid_object.best_params_
```

Visualizing the decision tree

One of the best aspects about building & implementing the decision tree in order to solve problems is that it can be interpreted quite easily by using a decision tree diagram that explains how the algorithm that you built works. In order to visualize a simple decision tree for the fraud detection dataset we use the code shown below:

```
#Package requirements

import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.externals.six import StringIO
```

```
from IPython.display import Image
from sklearn.tree import export_graphviz
import pydotplus
from sklearn import tree
```

We first start by importing the required packages. The new packages here are:

1. StringIO
2. Image
3. export_graphviz
4. pydotplus
5. tree

The installations of the packages have been covered in chapter 1.

Next, we read in the dataset & initialize a decision tree classifier as shown in the code below:

```
#Reading in the data

df = pd.read_csv('fraud_prediction.csv')
df = df.drop(['Unnamed: 0'], axis = 1)

#Creating the features

features = df.drop('isFraud', axis = 1).values
target = df['isFraud'].values

#Initializing the DT classifier

dt = DecisionTreeClassifier(criterion = 'gini', random_state = 50,
max_depth= 5)
```

We then fit the tree on the features and target and extract the feature names separately:

```
#Fitting the classifier on the data

dt.fit(features, target)

#Extracting the feature names

feature_names = df.drop('isFraud', axis = 1)
```

We can then visualize the decision tree by using the code shown below:

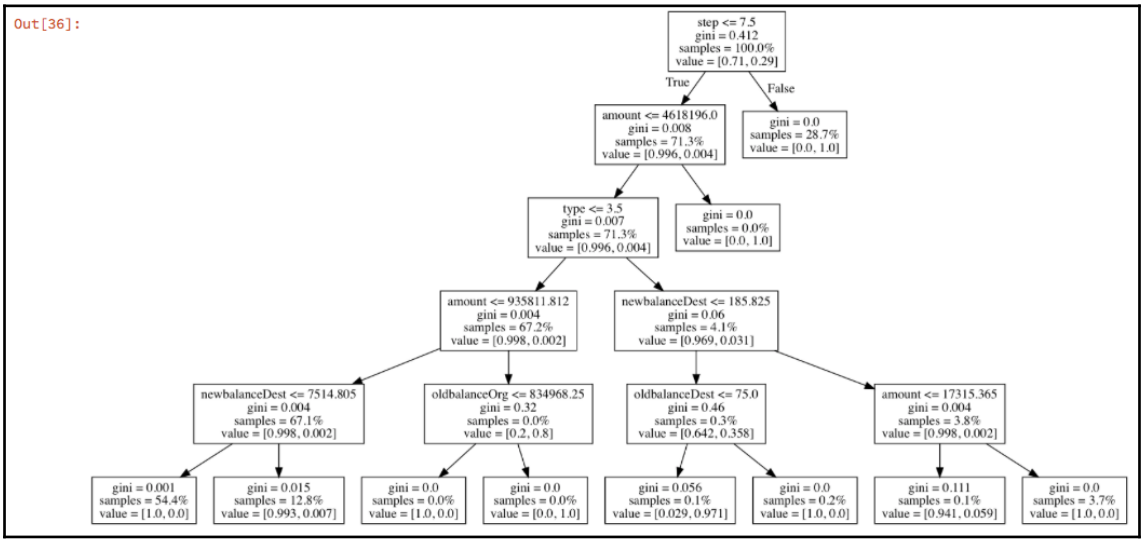
```
#Creating the tree visualization
```

```
data = tree.export_graphviz(dt, out_file=None, feature_names=
feature_names.columns.values, proportion= True)

graph = pydotplus.graph_from_dot_data(data)

# Show graph
Image(graph.create_png())
```

In the code above:

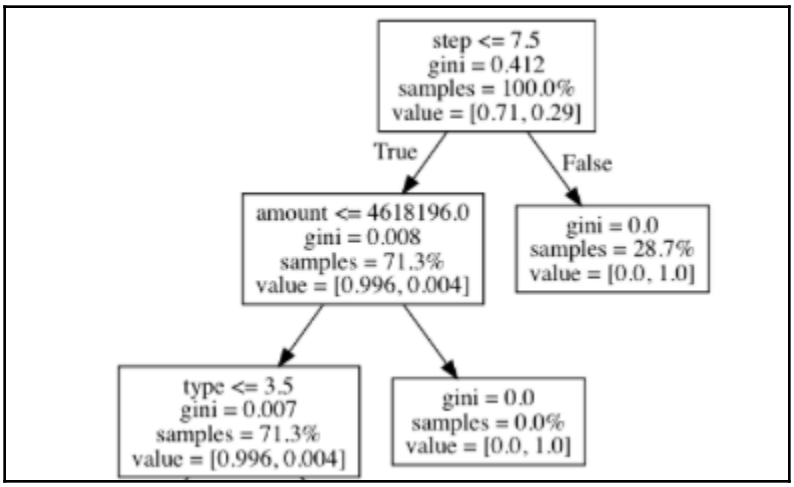
1. We use the *tree.export_graphviz()* function in order to construct the decision tree object and store it in a variable called "data".
2. This function uses a couple of arguments - The decision tree that you built "dt", the out_file is set to None as we do not want to send the tree visualization to any file outside our Jupyter Notebook, the feature names that we defined earlier and the proportion which is set to True (This will be explained later)
3. We then construct a graph of the data contained within the tree so that we can visualize this decision tree graph by using the *pydotplus.graph_from_dot_data()* function on the "data" variable which contains the data about the decision tree.
4. Finally, we visualize the decision tree by using the *Image()* function and passing the graph of the decision tree to it.

This results in a decision tree as illustrated in the image shown below:



The resulting decision tree

The tree might seem pretty complex to interpret at first, but it's not very hard to interpret. In order to interpret this tree, let's consider the root node and the first two children only. This is illustrated in the image below:



Snippet of the decision tree

In the tree above:

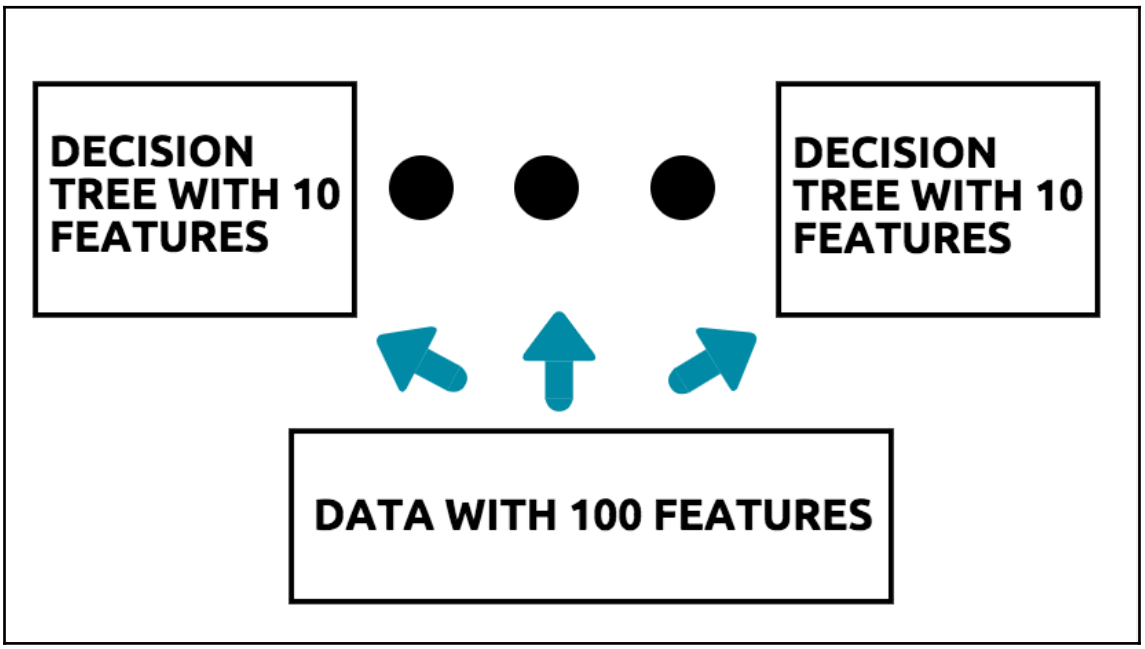
1. In the root node, the tree has identified the 'step' feature as the feature with the highest Gini value.
2. The root node makes the split in such a way that - 0.71 or 71% of the data falls into the non-fraudulent transactions while 0.29 or 29% of the transactions fall into the fraudulent transactions.
3. If the step is greater than or equal to 7.5 (The right side), then all of the transactions are classified as fraudulent.
4. If the step is lesser than or equal to 7.5 (The left side), then 0.996 or 99.6% of the transactions are classified as non-fraudulent while 0.004 or 0.4% of the transactions are classified as fraudulent.
5. If the amount is greater than or equal to 4618196.0 then all of the transactions are classified as fraudulent.
6. If the amount is lesser than or equal to 4618196.0 then 0.996 or 99.6% of the transactions are classified as non-fraudulent while 0.004 or 0.4% of the transactions are classified as fraudulent.

Notice, how the decision tree is simply a set of If-then rules constructed in a nested manner.

The Random Forests Classifier

Now that you have understood the core principles of the decision tree at a very foundational level we will now explore what the Random Forests are. Random Forests are a form of **ensemble** learning. An ensemble learning method is one which makes use of multiple machine learning models to make a decision.

Let's consider the image illustrated below:



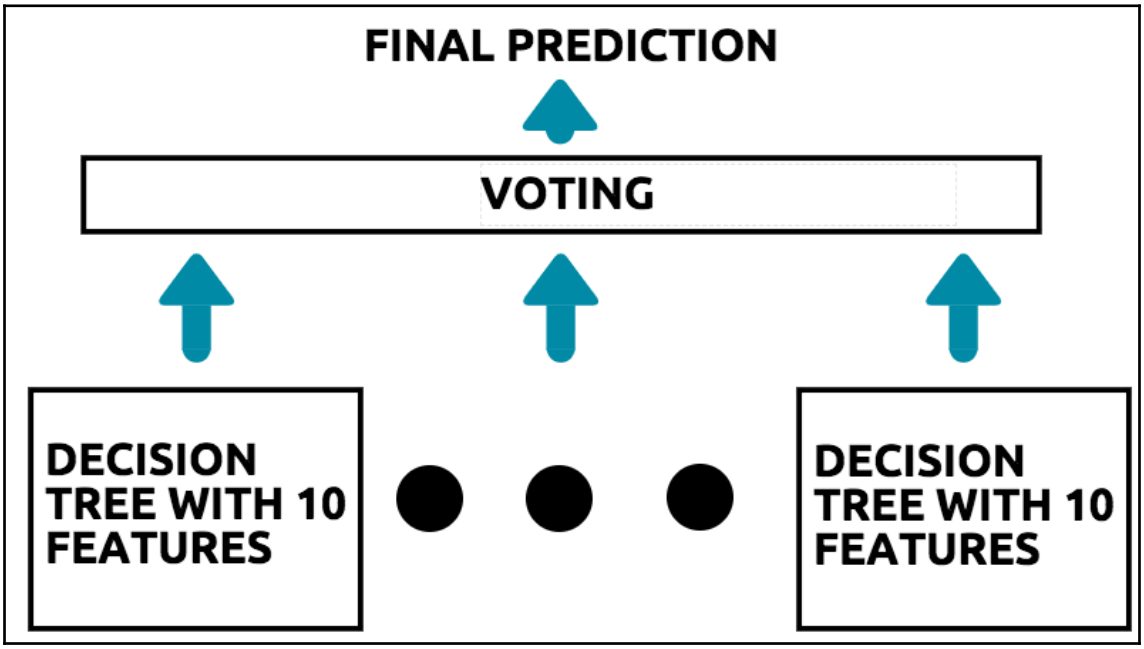
The concept of ensemble learning

In the Random Forest algorithm:

1. Assume that you initially have a dataset with a 100 features.
2. We will build a decision tree with 10 features initially. The features are selected randomly.
3. Now, with the remaining 90 features, we construct the next the decision tree with 10 features.
4. This process continues until there are no more features left to build a decision tree with.
5. At this point of time, we have 10 decision trees - each with 10 features.

6. Each decision tree is known as the **base estimator** of the Random Forest.
7. Thus we have a forest of trees each built using a random set of 10 features.

The next step for the algorithm is to make the prediction. In order to better understand how the Random Forest algorithm makes predictions consider the image shown below:



Process of making predictions in Random Forests

In the image above:

1. Assume that there are 10 decision trees in total in the Random Forest.
2. Each decision tree makes a single prediction for the data that comes in.
3. If 6 trees predicts class A and 4 trees predict class B then the final prediction of the Random Forest algorithm is class A as it had the majority vote.
4. This process of voting on a prediction based on the outputs of multiple models is known as ensemble learning.

Now that you have learnt how the algorithm works internally, we can now implement it using scikit-learn!

Implementing the Random Forest classifier in scikit-learn

In this section we will implement the Random Forest classifier in scikit-learn. The first step is to read in the data and split the data into training and test sets. This can be done by using the code shown below:

```
import pandas as pd

#Reading in the dataset

df = pd.read_csv('fraud_prediction.csv')

#Dropping the index

df = df.drop(['Unnamed: 0'], axis = 1)

#Creating the features

features = df.drop('isFraud', axis = 1).values
target = df['isFraud'].values

X_train, X_test, y_train, y_test = train_test_split(features, target,
test_size = 0.3, random_state = 42, stratify = target)
```

The next step is to build the Random Forest Classifier. We can do that by using the code shown below:

```
from sklearn.ensemble import RandomForestClassifier

#Initiliazing an Random Forest Classifier with default parameters

rf_classifier = RandomForestClassifier(random_state = 50)

#Fitting the classifier on the training data

rf_classifier.fit(X_train, y_train)

#Extracting the scores

rf_classifier.score(X_test, y_test)
```

In the code above:

1. We first import the *RandomForestClassifier* from scikit-learn.
2. Next, we initialize a Random Forest Classifier model.

3. We then fit this model to our training data & evaluate it's accuracy on the test data.

Hyper-parameter tuning for the Random Forest

In this section we will learn how to optimize the hyper-parameters of the Random Forest algorithm. Since the Random Forests are fundamentally based on multiple decision trees, the hyper-parameters are very similar. In order to optimize the hyper-parameters we use the code shown below:

```
from sklearn.model_selection import GridSearchCV

#Creating a grid of different hyper-parameters

grid_params = {
 'n_estimators': [100,200, 300,400,5000],
 'max_depth': [1,2,4,6,8],
 'min_samples_leaf': [0.05, 0.1, 0.2]
}

#Building a 3 fold Cross-Validated GridSearchCV object

grid_object = GridSearchCV(estimator = rf_classifier, param_grid =
grid_params, scoring = 'accuracy', cv = 3, n_jobs = -1)

#Fitting the grid to the training data

grid_object.fit(X_train, y_train)

#Extracting the best parameters

grid_object.best_params_

#Extracting the best model

rf_best = grid_object.best_estimator_
```

In the code above:

1. We first import the *GridSearchCV* package.
2. We initialize a dictionary of hyper-parameter values. The 'max_depth' & 'min_samples_leaf' are similar to that of the decision tree.
3. However, the 'n_estimators' is a new parameter which talks about the total number of trees that you want your Random Forest algorithm to consider while making the final prediction.

4. We then build and fit the grid search object to the training data and extract the best parameters.
5. The best model is then extracted using these optimal hyper-parameters.

The AdaBoost Classifier

In the section you will learn how the AdaBoost classifier works internally and how the concept of 'boosting' might give you better results. Boosting is a form of ensemble machine learning in which machine learning models learn from the mistakes of the models that were previously built, thereby increasing it's final prediction accuracy.

AdaBoost stands for Adaptive Boosting and is a boosting algorithm in which a lot of importance is given to rows of data that the initial predictive model got wrong. This ensures that the next predictive model will not make the same mistakes.

The process by which the AdaBoost algorithm works is illustrated below:

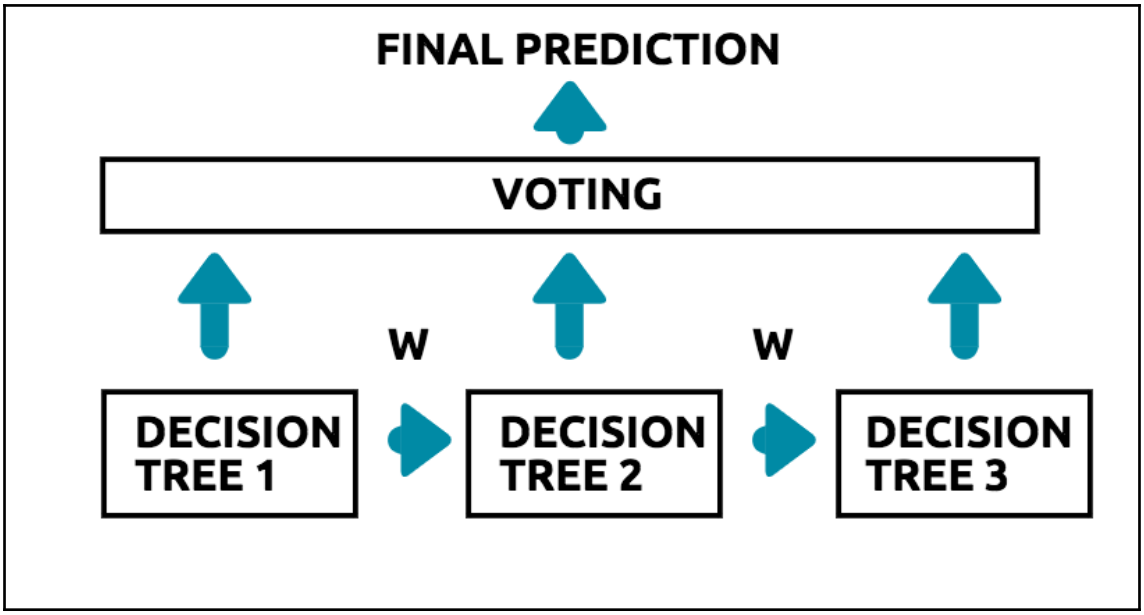


Illustration of the AdaBoost algorithm

In the picture of the AdaBoost algorithm illustrated above:

1. The first decision tree is built and outputs a set of predictions.

2. The predictions that the first decision tree got wrong is given a weight of "w". This means that if the weight is set to 2, then 2 instances of that particular sample are introduced into the dataset.
3. This enables decision tree 2 to learn better since we have more samples of the data that it made an error with in the first place.
4. This process is repeated until all the trees are built.
5. Finally, the predictions of all the trees are gathered and a weighted vote is initiated in order to determine the final prediction.

Implementing the AdaBoost classifier in scikit-learn

In this section we will learn how we can implement the AdaBoost classifier in scikit-learn in order to predict if a transaction is fraudulent or not. As usual, the first step is to import the data & split it into training and testing sets.

This can be done by using the code shown below:

```
#Reading in the dataset

df = pd.read_csv('fraud_prediction.csv')

#Dropping the index

df = df.drop(['Unnamed: 0'], axis = 1)

#Creating the features

features = df.drop('isFraud', axis = 1).values
target = df['isFraud'].values

X_train, X_test, y_train, y_test = train_test_split(features, target,
test_size = 0.3, random_state = 42, stratify = target)
```

The next step is to build the AdaBoost classifier. We can do this by using the code shown below:

```
from sklearn.ensemble import AdaBoostClassifier

#Initialize a tree (Decision Tree with max depth = 1)

tree = DecisionTreeClassifier(max_depth=1, random_state = 42)

#Initialize an AdaBoost classifier with the tree as the base estimator

ada_boost = AdaBoostClassifier(base_estimator = tree, n_estimators=100)
```

```
#Fitting the AdaBoost classifier to the training set

ada_boost.fit(X_train, y_train)

#Extracting the accuracy scores from the classifier

ada_boost.score(X_test, y_test)
```

In the code above:

1. We first import the *AdaBoostClassifier* package from scikit-learn.
2. Next, we initialize a decision tree that forms the base of our AdaBoost classifier.
3. We then build the AdaBoost classifier with the base estimator as the decision tree and we specify that we want 100 decision trees in total.
4. Finally, we fit the classifier to the training and extract the accuracy scores from the test data.

Hyper-parameter tuning for the AdaBoost classifier

In this section we will learn how to tune the hyper-parameters of the AdaBoost classifier. The AdaBoost classifier has only one parameter of interest which is the number of base estimators or decision tress.

We can optimize the hyper-parameters of the AdaBoost classifier by using the code shown below:

```
from sklearn.model_selection import GridSearchCV

#Creating a grid of hyper-parameters

grid_params = {
    'n_estimators': [100,200,300]
}

#Building a 3 fold CV GridSearchCV object

grid_object = GridSearchCV(estimator = ada_boost, param_grid = grid_params,
scoring = 'accuracy', cv = 3, n_jobs = -1)

#Fitting the grid to the training data

grid_object.fit(X_train, y_train)

#Extracting the best parameters
```

```
grid_object.best_params

#Extracting the best model

ada_best = grid_object.best_estimator_
```

In the code above:

1. We first import the *GridSearchCV* package.
2. We initialize a dictionary of hyper-parameter values. In this case, the 'n_estimators' is the number of decision trees.
3. We then build and fit the grid search object to the training data and extract the best parameters.
4. The best model is then extracted using these optimal hyper-parameters.

Regression Trees

You have learnt how trees are used in order to classify a prediction as belonging to particular class or category. However, trees can also be used to solve problems related to predicting numeric outcomes. In this section you will about the three types of tree based algorithms that you can implement in scikit-learn in order to predict numeric outcomes instead of classes:

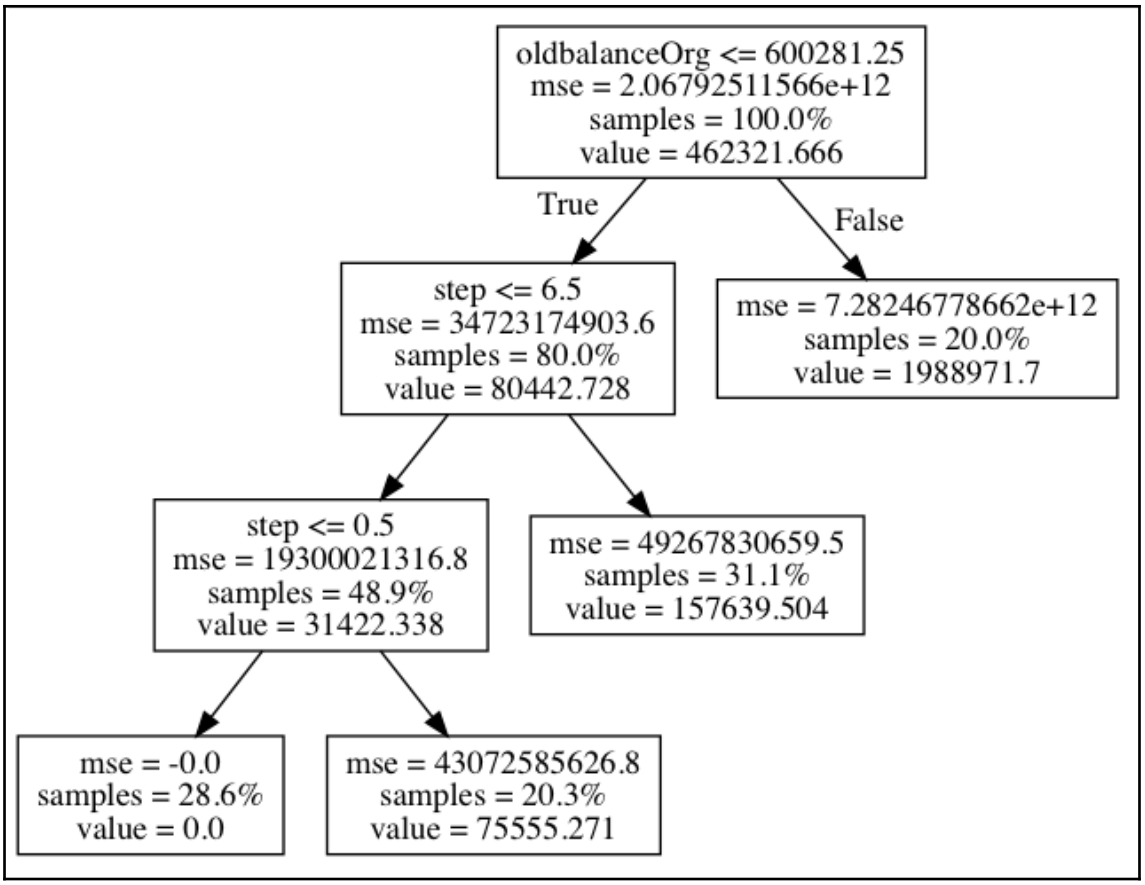
1. The Decision Tree Regressor
2. The Random Forest Regressor
3. The Gradient Boosted Tree

The Decision Tree Regressor

When we have data that is non-linear in nature, a linear regression model might not be the best model to choose. In such situations it makes sense to choose a model that can fully capture this non-linearity in the data. A decision tree regressor can be used to predict numeric outcomes just like that of the linear regression model.

In the case of the Decision Tree Regressor, we use the Mean Squared Error instead of the Gini metric in order to determine how the tree is built. You will learn about the Mean Squared Error in detail in chapter 8 - Performance Evaluation Methods. In a nutshell, the Mean Squared Error or the MSE is used to tell us about the error in prediction.

Consider the tree shown in the image below:



Decision Tree for Regression

In the image of the Decision Tree above:

1. We are trying to predict the amount of a mobile transaction using the tree.
2. When the tree tries to decide on a split, it chooses the node in such a way that the target values (amount) is closest to the mean values of the target in that node.
3. You will notice that as you go down the tree to the left (The True Cases), the Mean Squared Error of the nodes decreases.
4. Therefore, the nodes are built in a recursive fashion such that it reduces the overall Mean Squared Error, therefore obtaining the 'true' value.
5. In the tree above, if the Old Balance of Origination is lesser than 600,281, then the amount is 80,442 and if it's greater than 600,281 then the amount is 1988971.
6. The amount is displayed by the parameter "value" in the tree.

Implementing the decision tree regressor in scikit-learn

In this section you will learn how to implement the decision tree regressor in scikit-learn. The first step is to import the data & create the features and target variables. We can do this by using the code shown below:

```
import pandas as pd

#Reading in the dataset

df = pd.read_csv('fraud_prediction.csv')

#Dropping the index

df = df.drop(['Unnamed: 0'], axis = 1)

#Creating the features

features = df.drop('amount', axis = 1).values
target = df['amount'].values
```

Note how in the case of regression the target variable is the amount and not the 'isFraud' column.

Next, we split the data into training and test sets and build the decision tree regressor as shown below:

```
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor

#Splitting the data into training and test sets

X_train, X_test, y_train, y_test = train_test_split(features, target,
test_size = 0.3, random_state = 42)

#Building the decision tree regressor

dt_reg = DecisionTreeRegressor(max_depth = 10, min_samples_leaf = 0.2,
random_state= 50)

#Fitting the tree to the training data

dt_reg.fit(X_train, y_train)
```

In the code above:

1. We first import the required packages & split the data into training and test sets.

2. Next, we build the decision tree regressor by using the *DecisionTreeRegressor()* function.
3. We specify two hyper-parameter arguments - The 'max_depth' which tells the algorithm how many branches the tree must have, the 'min_sample_leaf' which tells the tree about the minimum number of samples that each node must have which is set to 20% or 0.2 of the total data in this case.
4. The 'random_state' is set to 50 to ensure that the same tree is built every time we run the code.
5. We then fit the tree to the training data.

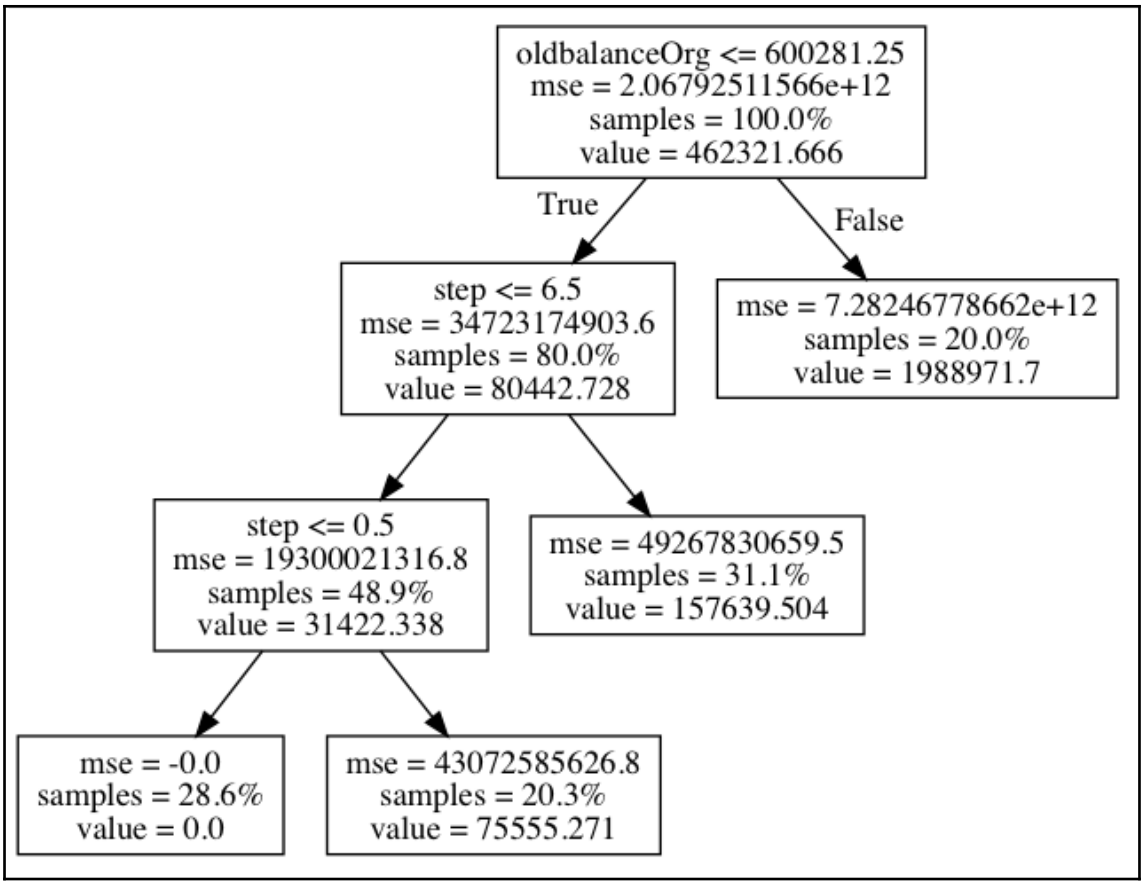
Visualizing the decision tree regressor

Just like how we visualized the decision tree classifier we can also visualize the decision tree regressor. Instead of showing you the classes/categories to which the node of a tree belong to, you will now be shown the value of the target variable.

We can visualize the decision tree regressor by using the code shown below:

```
#Package requirements

from sklearn.tree import DecisionTreeClassifier
from sklearn.externals.six import StringIO
from IPython.display import Image
from sklearn.tree import export_graphviz
import pydotplus
from sklearn import tree

#Extracting the feature names

feature_names = df.drop('amount', axis = 1)

#Creating the tree visualization

data = tree.export_graphviz(dt_reg, out_file=None, feature_names=
feature_names.columns.values, proportion= True)

graph = pydotplus.graph_from_dot_data(data)

# Show graph
Image(graph.create_png())
```

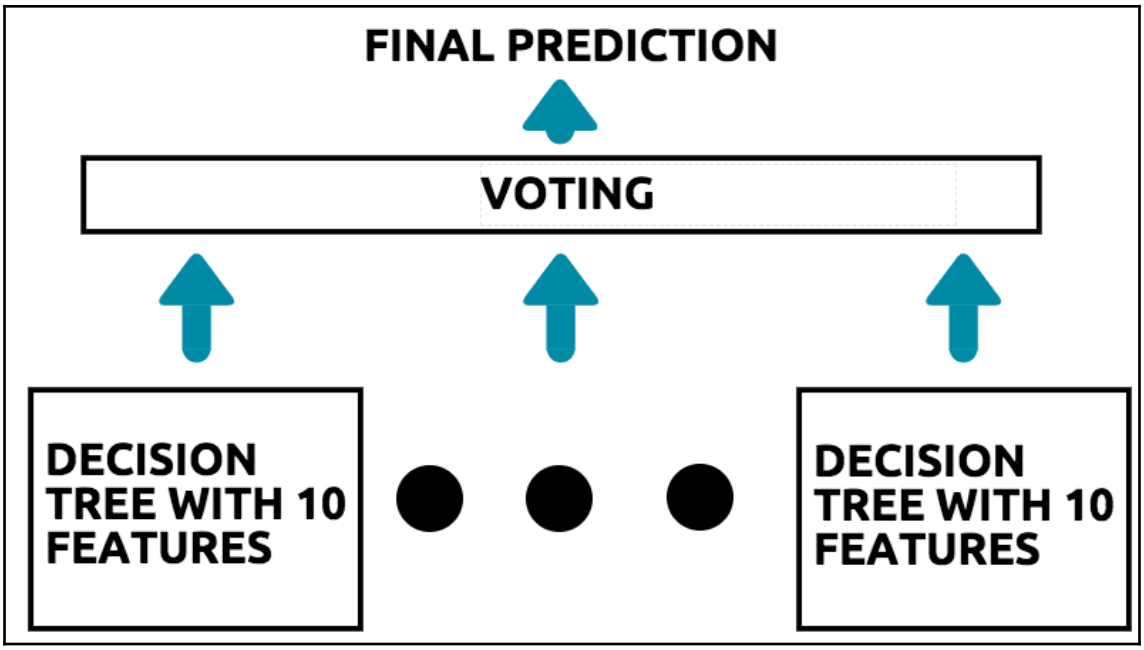
The code follows the exact same methodology as in the case of the decision tree classifier and will not be discussed in detail here. This produces a decision tree regressor as illustrated in the image below:



The Decision Tree Regressor visualized.

The Random Forest Regressor

The Random Forest Regressor takes the Decision Tree Regressor as the base estimator and makes predictions in a method similar to that of the Random Forest Classifier as illustrated in the image below:



Making the final prediction in the Random Forest Regressor

The only difference between the Random Forest Classifier and the Random Forest Regressor is the fact that the base estimator is a decision tree regressor in the case of the latter.

Implementing the Random Forest Regressor in scikit-learn

In this section you will learn how you can implement the Random Forest Regressor in scikit-learn. The first step is to import the data and split it into training & testing sets. This can be done using the code shown below:

```
import pandas as pd
from sklearn.model_selection import train_test_split

#Reading in the dataset

df = pd.read_csv('fraud_prediction.csv')

#Dropping the index
```

```
df = df.drop(['Unnamed: 0'], axis = 1)

#Creating the features & target arrays

features = df.drop('amount', axis = 1).values
target = df['amount'].values

#Splitting the data into training and test sets

X_train, X_test, y_train, y_test = train_test_split(features, target,
test_size = 0.3, random_state = 42)
```

The next step is to build the Random Forest Regressor. We can do this using the code shown below:

```
from sklearn.ensemble import RandomForestRegressor

#Initiliazing an Random Forest Regressor with default parameters

rf_reg = RandomForestRegressor(max_depth = 10, min_samples_leaf = 0.2,
random_state = 50)

#Fitting the regressor on the training data

rf_reg.fit(X_train, y_train)
```

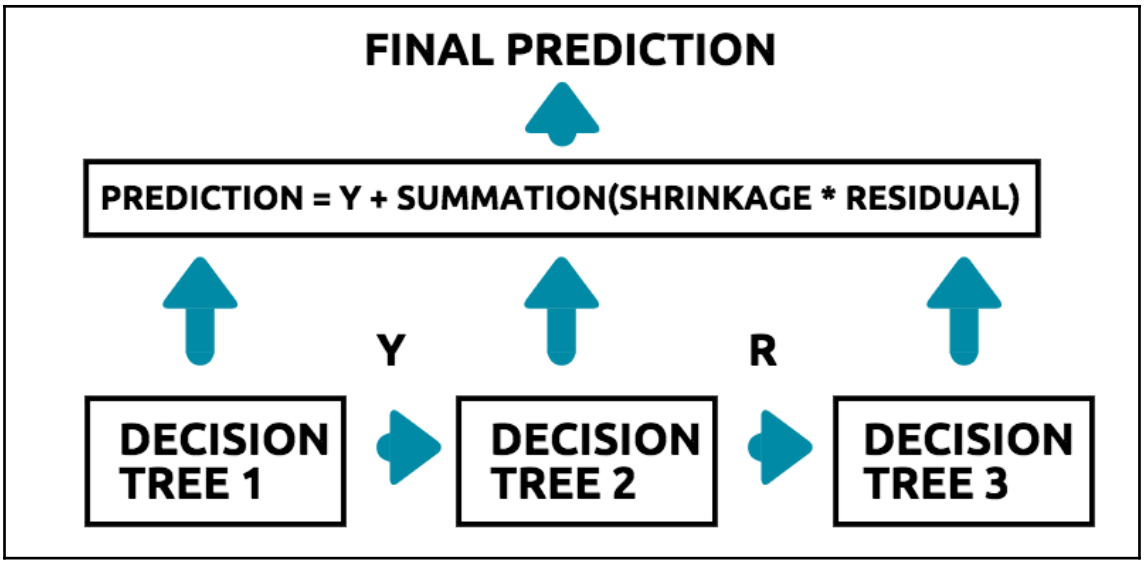
In the code above:

1. We first import the *RandomForestRegressor* module from scikit-learn.
2. We then initialize a Random Forest Regressor object called "rf_reg" with a max depth of each decision tree as 10 and the minimum number of data/samples in each tree as 20% of the total data.
3. We then fit the tree to the training set.

The Gradient Boosted Tree

In this section you will learn how the gradient boosted tree is used for regression & how you can implement the same using scikit-learn.

In the AdaBoost Classifier that you have learnt about earlier in this chapter, weights are added to the examples that the classifier predicted wrong. In the gradient boosted tree however, instead of weights the residual errors are used as labels in each tree in order to make future predictions. This concept is illustrated for you in the image shown below:



In the image above:

1. The first decision tree is trained with the data that you have and the target variable Y.
2. We then compute the residual error for this tree.
3. The residual error is given by the difference between the predicted value and the actual value.
4. The second tree is now trained by using the residuals as the target.
5. This process of building multiple trees is iterative and continues for the number of base estimators that we have.
6. The final prediction is = The target value predicted by the first tree + the product of the shrinkage and the residuals for all the other trees.
7. The shrinkage is a factor by which we control the rate at which we want this process gradient boosting to take place.
8. A small value of shrinkage (learning rate) implies that the algorithm will learn fast and therefore must be compensated with a larger number of base estimators (decision trees) in order to prevent overfitting.
9. A larger value of shrinkage (learning rate) implies that the algorithm with learn slower and thus requires fewer trees in order to reduce the computational time.

Implementing the gradient boosted tree in scikit-learn

In this section we will learn how we can implement the gradient boosted regressor in scikit-learn. The first step as usual is to import the dataset, define the features and target arrays and split the data into training and test sets. This can be done using the code shown below:

```
import pandas as pd
from sklearn.model_selection import train_test_split

#Reading in the dataset

df = pd.read_csv('fraud_prediction.csv')

#Dropping the index

df = df.drop(['Unnamed: 0'], axis = 1)

#Creating the features

features = df.drop('amount', axis = 1).values
target = df['amount'].values

#Splitting the data into training and test sets

X_train, X_test, y_train, y_test = train_test_split(features, target,
test_size = 0.3, random_state = 42)
```

The next step is to build the Gradient Boosted Regressor. This can be done using the code shown below:

```
from sklearn.ensemble import GradientBoostingRegressor

#Initializing an Gradient Boosted Regressor with default parameters

gb_reg = GradientBoostingRegressor(max_depth = 5, n_estimators = 100,
learning_rate = 0.1, random_state = 50)

#Fitting the regressor on the training data

gb_reg.fit(X_train, y_train)
```

In the code above:

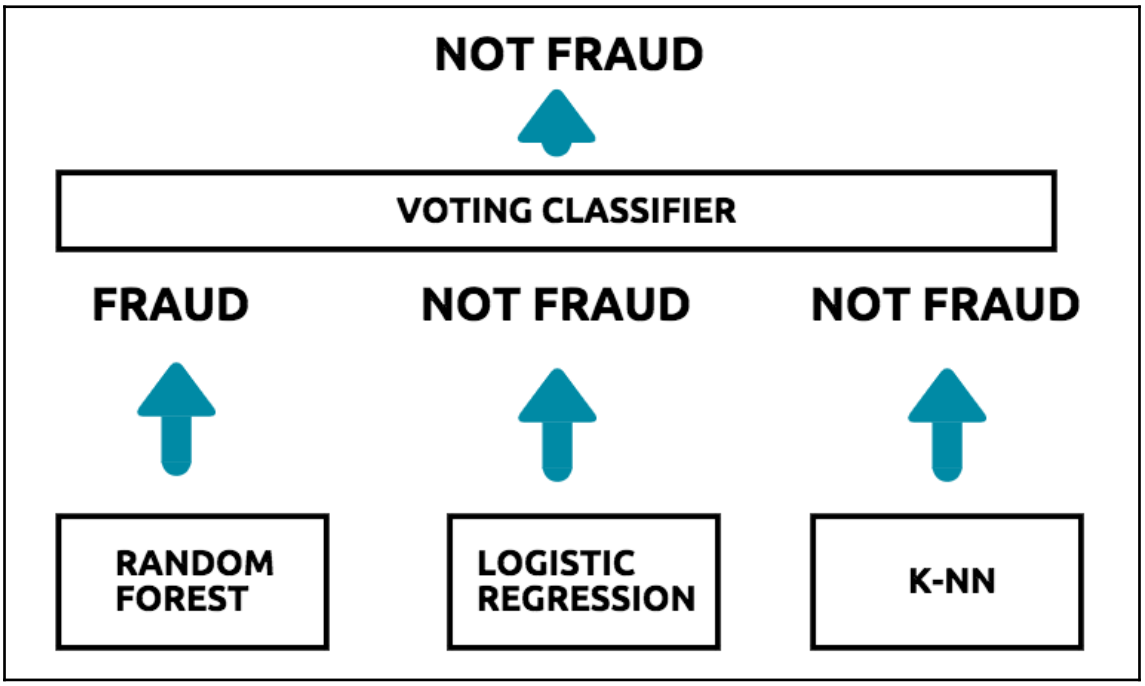
1. We first import the *GradientBoostingRegressor* from scikit-learn.
2. We the build a Gradient Boosted Regressor object with three main arguments - The maximum depth of each tree, the total number of trees & the learning rate.
3. We then fit the regressor on the training data.

Ensemble Classifier

The concept of ensemble learning was explored throughout the entirety of the chapter when we learned about Random Forests, AdaBoost & Gradient Boosted Trees. However, this concept can be extended into classifiers outside of trees.

If we had built a logistic regression, Random Forest & K-Nearest Neighbors classifiers & we wanted to group them all together and extract the final prediction through majority voting - this can be done by using the Ensemble classifier.

This concept can be better understood with a diagram as illustrated in the image below:



Ensemble Learning with a Voting Classifier to predict Fraud Transactions

In the diagram above:

1. The Random Forest classifier predicted that a particular transaction was a fraud while the other two transactions predicted that the transaction was not fraud.
2. The Voting Classifier sees that 2 out of 3 (majority) of the predictions are 'Not Fraud' and hence outputs the final prediction as 'Not Fraud'.

Implementing the Voting Classifier in scikit-learn

In this section you will learn how to implement the voting classifier in scikit-learn. The first step is to import the data, create the feature and target arrays and create the training and testing splits. This can be done using the code shown below:

```
import pandas as pd
from sklearn.model_selection import train_test_split

#Reading in the dataset

df = pd.read_csv('fraud_prediction.csv')

#Dropping the index

df = df.drop(['Unnamed: 0'], axis = 1)

#Splitting the data into training and test sets

X_train, X_test, y_train, y_test = train_test_split(features, target,
test_size = 0.3, random_state = 42)
```

Next, we will build two classifiers to include in the Voting Classifier - The Decision Tree Classifier & the Random Forest Classifier. This can be done using the code shown below:

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier

#Initializing the DT classifier

dt = DecisionTreeClassifier(criterion = 'gini', random_state = 50)

#Fitting on the training data

dt.fit(X_train, y_train)

#Initiliazing an Random Forest Classifier with default parameters

rf_classifier = RandomForestClassifier(random_state = 50)

#Fitting the classifier on the training data

rf_classifier.fit(X_train, y_train)
```

Next, we will build the Voting Classifier by using the code shown below:

```
from sklearn.ensemble import VotingClassifier
```

```
#Creating a list of models

models = [('Decision Tree', dt), ('Random Forest', rf_classifier)]

#Initialize a voting classifier

voting_model = VotingClassifier(estimators = models)

#Fitting the model to the training data

voting_model.fit(X_train, y_train)

#Evaluating the accuracy on the test data

voting_model.score(X_test, y_test)
```

In the code above:

1. We first import the *VotingClassifier* module from scikit-learn.
2. Next we create a list of all the models that we want to use in our Voting Classifier.
3. In the list of classifiers, each model is stored in a tuple with the model's name in a string and the model itself.
4. We then initialize a Voting Classifier with the list of models we built in step 2.
5. Finally, the model is fit to the training data and the accuracy is extracted from the test data.

Summary

While this chapter was rather long, you have entered the world of tree based algorithms & left with a wide arsenal of tools that you can implement in order to solve both small & large scale problems. To summarize, you have learnt:

1. Decision Trees for classification & regression
2. Random Forests for classification & regression
3. AdaBoost for classification
4. Gradient Boosted Trees for regression
5. The Voting Classifier that can be used to build a single model out of different models.

In the upcoming chapter you will learn how you can work with data that does not have a target variable/labels and perform unsupervised machine learning in order to solve such

problems!

Index