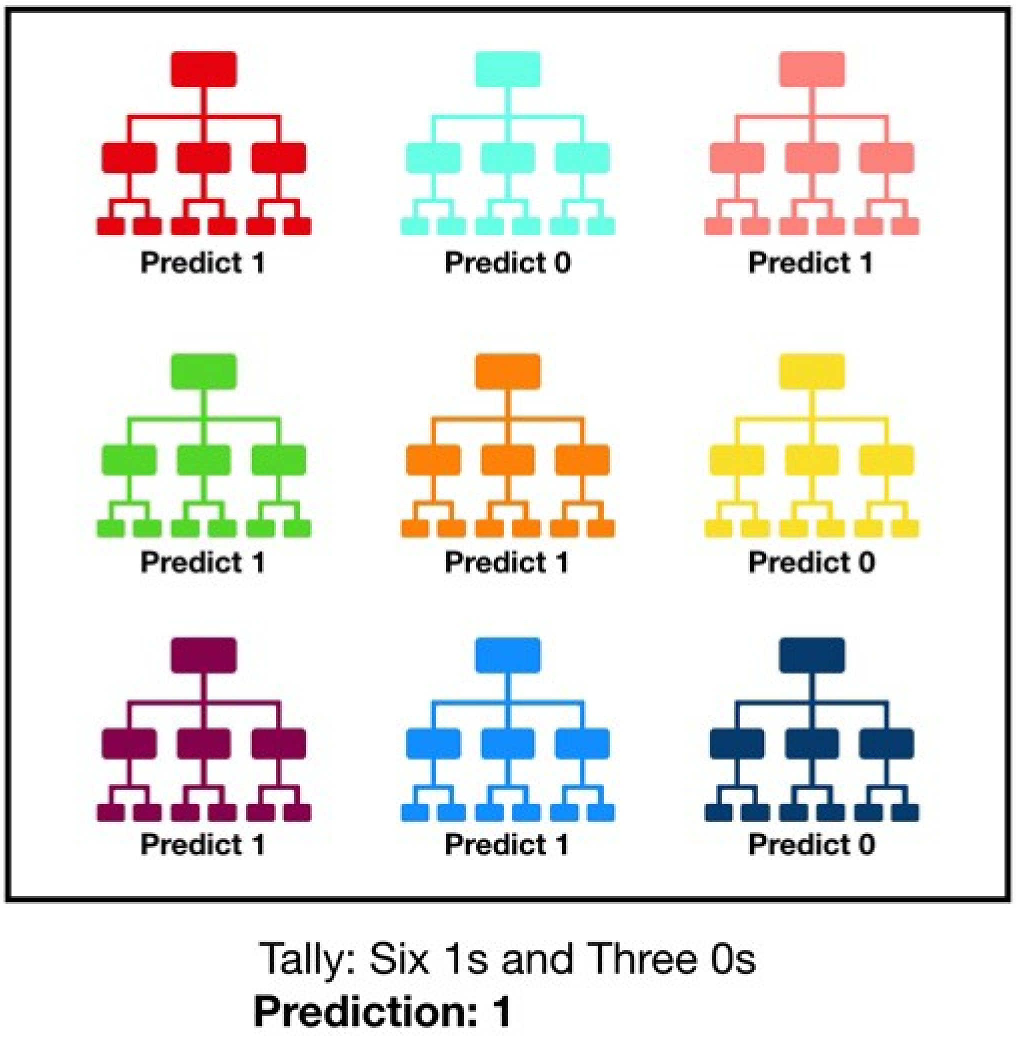
Random Forests

This lesson will focus on training random forest models in Python.

WE'LL COVER THE FOLLOWING

Random Forests

Random Forest, as the name implies, consists of a large number of decision trees that operate as an *ensemble* for classification. An **ensemble** is a collection of different predictive models that collectively decide the predicted output. In random forests, each individual tree gives a class as an output. The class with the most votes gets chosen as the final output of the model.



Picture courtesy of towardsdatascience.com

In the above example, we can see that six trees predict class `1` while three predict `0`. Class `1` has more votes; hence the final prediction is `1`.

How do Random Forests work?

The idea behind random forests is that a large number of *uncorrelated* decision trees working individually will perform better as a *committee* than any individual tree.

There are two important words in the above statement.

- Uncorrelated trees
- Performing as a committee

Uncorrelated trees

For a random forest model to perform nicely, the individual decision tree models need to have low correlation amongst themselves. Just like investments with low correlations, such as stocks and bonds, combine to form a portfolio greater than the sum of its parts, a random forest can produce predictions better than its individual trees. The reason is that all trees do not have the same error. Rather each tree has a different kind of error. They collectively move in a direction to reduce the total error. Therefore, the predictions made by individual trees need to have a low correlation.

To ensure that the outcomes of the individual decision trees are uncorrelated, random forests use two techniques:

- Bagging
- Feature randomness

Bagging

Decision trees are very sensitive to training data, and any change in training data can significantly change the outcome of the model. Random forests make use of this observation. Each individual decision tree randomly samples from the training data with replacement. This means each tree has a different training set, which leads to different decision tree models. This technique of using random samples with replacement is known as **bagging**.

Note that we do not train individual trees on random subsets of the data, rather they are trained on the whole data set where each training example is randomly sampled with replacement. For instance, if our training data has 6 observations such as [1,2,3,4,5,6] and we sample 6 times with replacement, we might get [1,2,2,4,5,5]. Therefore, each individual tree will have a different training set with some overlap.

Therefore, each individual decision tree is different.

Feature randomness

In decision trees, when we split a node, we consider all features and then decide on the feature that gives us the most separation between the left node and the right node. But in random forests, individual trees can only select from a subset of features. This introduces more variability among the trees. Therefore, the trees made have low correlation amongst themselves.

So, in random forest models, we produce individual decision trees that are not only trained on different sets of data (thanks to bagging) but also use different features to make decisions.

Performing as a committee

In the end, a random forest model chooses the class which had the most votes from individual decision trees.

Random Forests in Python

We can easily use random forests for our predictions in Python. The model is available in `sklearn.ensemble` as `RandomForestClassifier`. We will be using the Default of Credit Card Clients dataset to make our predictions.

```python
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score,classification_report

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

# MAKE DATA
X = df.drop(columns = ['default.payment.next.month','MARRIAGE','GENDER'])
Y = df[['default.payment.next.month']]

X_train,X_test,Y_train,Y_test = train_test_split(X,Y,test_size = 0.2, random_state = 3)

# MAKE MODEL
forest = RandomForestClassifier()
forest.fit(X_train,Y_train)

# CALCULATE AND PRINT RESULTS
preds = forest.predict(X_test)
acc = accuracy_score(y_true = Y_test,y_pred = preds)
print(acc)
print(classification_report(y_true = Y_test,y_pred = preds))
```

We import the `RandomForestClassifier` in **line 2**. After we read the data in **line 6**, we separate our target variable as `Y`. To split the data into training and test sets, we use the function `train_test_split`. We provide our inputs `X` and the labels `Y` to the function in **line 12**. We also provide the test set size as `test_size`. 0.2 implies that 20% data will be included in the testing set, while the remaining 80% will form the training set. The function outputs 4 items that we can retrieve directly into 4 variables. These are:

1. Inputs for the training data that we store in `X_train`

2. Inputs for the testing data that we store in `X_test`
3. Labels for the training data that we store in `Y_train`
4. Labels for the testing data that we store in `Y_test`

In **line 15**, we make our Random Forest classifier object just like we did in the last lesson. We call `RandomForestClassifier` without any arguments. Then in the next line, we call the `fit` function of the model. We provide the training examples and labels to the function. After this, we need to evaluate our model. Therefore, we use the `predict` function of the model in **line 19** to store predictions in `preds`. We give the testing inputs `X_test` to `predict` as an argument. We use `accuracy_score` function to measure the accuracy of the predictions. We print the accuracy with the classification report, which we obtained by using the `classification_report` function, in the last two lines.

We see that the model is approximately 80% accurate.

Hyper parameters tuning

When we made the random forest model above, we did not mention how many trees to include in the model or how many features each tree should use. Parameters like these are called **hyperparameters** of a model.

Let's make a model by including 20 trees in the model. By default, it used 10 individual decision trees.

```
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score,classification_report

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

# MAKE DATA
X = df.drop(columns = ['default.payment.next.month','MARRIAGE','GENDER'])
Y = df[['default.payment.next.month']]

X_train,X_test,Y_train,Y_test = train_test_split(X,Y,test_size = 0.2, random_state = 3)

# MAKE MODEL
forest = RandomForestClassifier(n_estimators = 20)
forest.fit(X_train,Y_train)

# CALCULATE AND PRINT RESULTS
preds = forest.predict(X_test)
```

```
acc = accuracy_score(y_true = Y_test,y_pred = preds)
print(acc)


print(classification_report(y_true = Y_test,y_pred = preds))
```

All of the code is the same except **line 15** in which we give `n_estimator = 20`. This implies that the model should use 20 decision trees. We can see that the accuracy changes a bit. For a list of all hyperparameters, refer to the documentation of Random Forest Classifier.

In the next lesson, we will learn another machine learning model known as *Support Vector Machine*.