TEKSANDS

Random Forest

UNDERSTANDING CLASSIFICATION USING RANDOM FOREST
-- MACHINE LEARNING --

This is a free eBook from Teksands

www.teksands.ai | info@teksands.ai | +91 6362 732 428

Teksands is a Deep Tech Learning company dedicated towards creating new age Digital professionals. Our core strength is our Faculty Members who are Research Scholars in the respective fields in top institutes like IIT and NITs.

We invite you to our world. We have courses for budding as well as experienced Deep Tech Professionals.
Apart from that, Teksands is dedicated to serve the IT profession – we have a huge repository of Deep Tech Articles in our website which anyone can access free.

We bring you free eBooks regularly to help with your quest of Tech knowledge. We thank you for your interest in us.

Our courses:
https://teksands.ai/courses/

Our Knowledge Repository:
https://teksands.ai/article/

Table of Contents

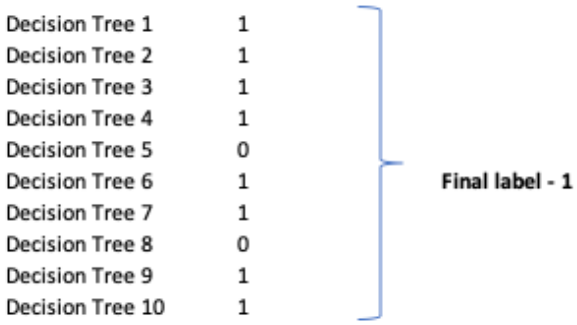
Random Forest

Random Forest is a Supervised Learning Algorithm that is essentially an Ensemble of randomly created Uncorrelated Decision Trees. Random Forest can be used in Classification as well as in Regression. For this discussion, we will limit ourselves to Classification only.

Let's look at some of the terms that are used in the definition.

Ensemble Methods: Ensemble methods are groups or ensembles of learning algorithms that collectively produce a result that is better than the strongest algorithm in the collective. The way this is achieved is that, after the individual learning algorithms produce the classification results during the training, the label that is produced by the most number of learners is taken as the final result of the classification for each individual observations. This is essentially a voting mechanism that uses the 'wisdom of crowd'.

In Random Forest, an Ensemble of Decision Trees are created that produce results (classification labels) on each observation and the classification results of each individual observation are finalized through a voting system as mentioned above. Let's take the following example of results on classification on an observation from a Random Forest of 10 trees.



In this scenario, we have a Random Forest with 10 Decision Trees and each of them have classified an observation as 1 or 0. 8 of them have classified it as 1 and two as 0.

Using Ensemble's voting technique, we settle for the final label of 1.

The above shows how voting would increase the probability of successfully classifying an observation.

Uncorrelated Decision Trees: One of the criteria for the Random Forest Ensemble of Decision Trees to be effective is that the individual trees have to be uncorrelated. Any correlation between multiple trees will increase the error rate of the Random Forest Classifier. Random Forest Learner takes in a random subset of features for the training. The smaller the subset is compared to the total number of features, the lesser the correlation and lesser the error rate.

Diversity: Random forest also ensures that the Trees in the Ensemble are as diverse as possible so that the

Some other key Features of Random Forest that makes it a preferred choice for many Classification use cases:

- Efficiently on large datasets
- Can handle large features sets
- Can provide details on Feature importance
- Can handle missing data, can work with features with missing data
- Can work well with unbalanced data sets

- Can work well with outlier data

The key differences between a normal Decision Tree model and Random Forest model are

- Decision Tree is one tree, Random Forest is an Ensemble of Trees
- In Random Forest, the trees ingest a subset of the data from the Input dataset
- Each Tree in a Random Forest takes in a subset of the Features so that every tree in the forest is different. This is to introduce diversity.

However, each tree (albeit with a subset of data and features) will be created much the same manner as a normal Decision Tree. Gini Index or Entropy are the measures that will be considered in node branching decisions. Also, the same Hyperparameters that are used to drive accuracy and generalisability of the model are used in Decision Trees and Random Forest. These are criterion, max_features, max_depth, min_samples_split, min_samples_leaf, min_weight_fraction_leaf, max_leaf_nodes, min_impurity_split, etc.

Common Use Cases

Random Forest can be used in a variety of real life Classification problems. Some of them are listed as below:

- Financial Services: Credit Card Fraud detection or Credit Default prediction
- E-commerce: Whether a customer is likely to shop for a particular product or type
- Cyber Security: Detecting whether an incoming request is an intrusion or legitimate one. Detecting Spam messages.
- Medicine: Predicting drug sensitivity. Predicting future conditions from current health conditions.

Note: The above use cases are Classification ones. We limit this article to Classification. Please note that Random Forest can also be used in Regression. That would be discussed in another article.

Other Classification Algorithms

For academic purpose, here is a list of competing Classification Algorithms that can also be used in similar problem areas. Each use case will come with a different input dataset and different requirement and challenges. Based on that, the choice of the Classification needs to be made. One of the options to chose is to apply multiple algorithms on a problem and compare the accuracy results.

- Naïve Bayes
- Support Vector Machine (SVM)
- Decision Tree
- K-Nearest Neighbour (KNN)
- Logistic Regression

Choosing the right Algorithm

Two metrices that are useful in comparing which Algorithm you should go for are, Accuracy and F1-Score.

Accuracy: Calculated as **(True Positive + True Negative) / Total Population.** This is essentially the ratio of correctly predicted observations against the total number of observations. Here, 'True Positive' is the number of correct predictions that the occurrence is positive and 'True Negative' is the number of correct predictions that the occurrence is negative. The higher the Accuracy of the model, the better.

F1-Score: This is the other metric that are used widely. F1-Score is calculated as F1-Score: **(2 x Precision x Recall) / (Precision + Recall)**. This score factors in both false positives and false negatives into account. F1-Score is usually more useful than accuracy, especially if you have an unbalanced dataset (wherein the proportion of one class is tiny in comparison to the other – e.g. 2 frauds detected in a population of 10000 Credit Card transactions). Here, 'Precision' is (True Positives / (True Positives + False Positive) and 'Recall' is True Positives / (True Positives + False Negatives).

Steps in a Random Forest Model building

The following figures details out the steps involved in a Random Forest model training process.

Step 1: Random sample of features are taken from the Input Dataset for each tree to be created. The number of features to be determined by 'n_features' parameter with the 'make_classification' method that can be applied on the X, y (input) datasets. The value of n_features should be less than total number of features in 'y'. it should be small enough to reduce overlap and increase tree diversity.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_classification
X, y = make_classification(n_samples=1000, n_features=4,
                           n_informative=2, n_redundant=0,
                           random_state=0, shuffle=False)
clf = RandomForestClassifier(max_depth=2, random_state=0)
clf.fit(X, y)
RandomForestClassifier(...)
print(clf.predict([[0, 0, 0, 0]]))
```

Step 2: Bootstrap a percentage of the Input data (typically two thirds) and feed into each tree in the forest. Remaining one thirds of the data will be used for validation as part of the tree building process. For the Bootstrapping and validation to happen, the parameters 'bootstrap' and 'oob_score' should be set as 'True'.

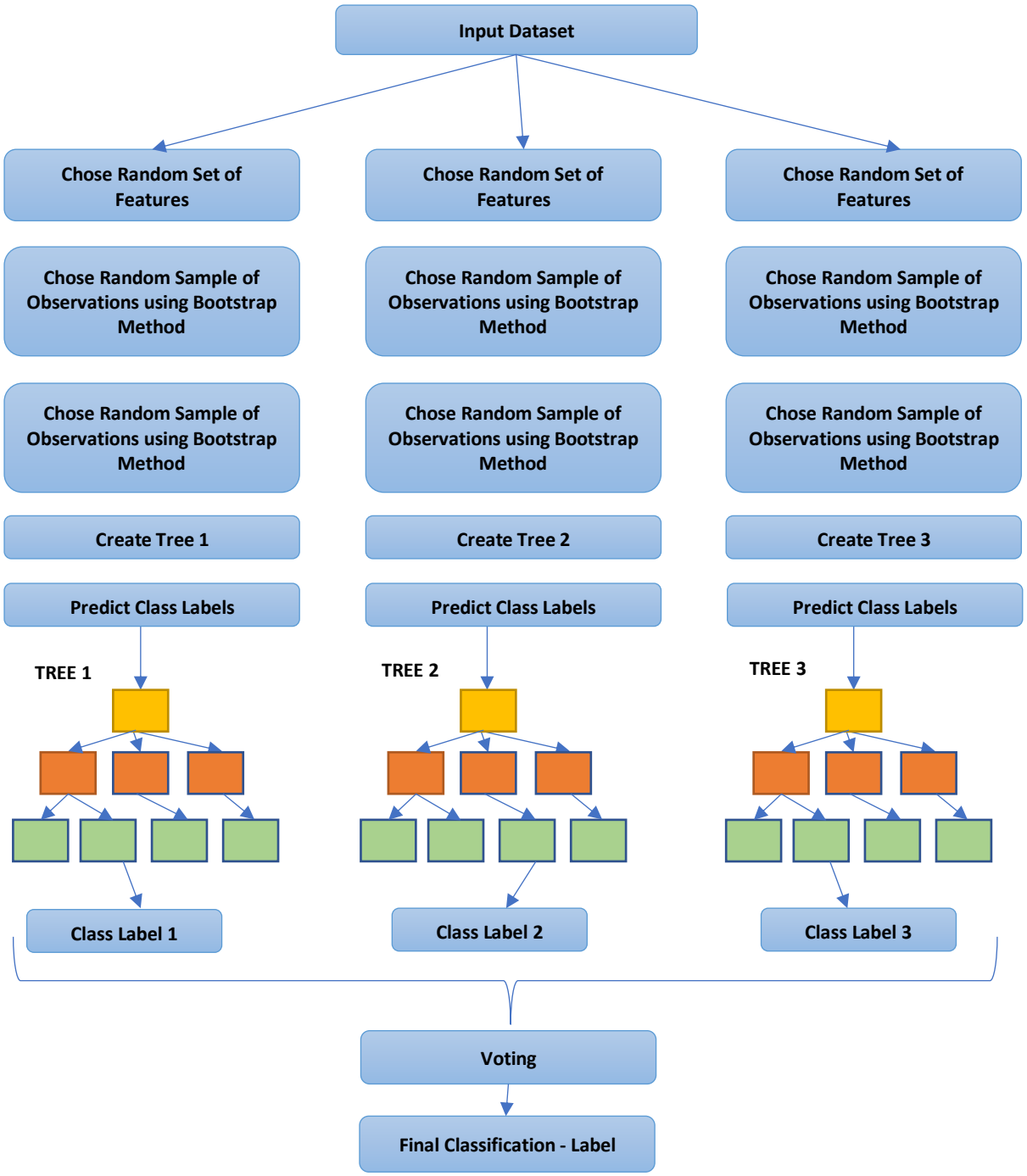
Note that 'n_estimators' parameter value should be set as the number of Trees that the model should produce. The more is better.

Step 3: The Trees in the Random Forest are created. The Node splitting should follow the same process as Decision Trees. At every node, split on the basis of the features and condition that would lead to the optimum value of 'gini index' or 'entropy', i.e. leading to the maximum homogeneity in the resultant nodes.

Step 4: Every tree in the forest will produce a classification.

Step 5: A voting process will look into the classification results from each tree for each observation and finalize the Classification based on 'Majority Voting'. The final class for any observation will be the one that is produced by most of trees.

Step 6: You can finally print the Accuracy / F1_score and oob_score for an estimate of the Model accuracy.



Out-of-Bag Error

The Accuracy of classifiers is a measure of the number of correctly classified observations divided by the total number of observations in the input dataset. We can, however, get a better approximation of the expected error from our Random Forest model through the Out of Bag (OOB) method it uses.

In random forests, there is no need for cross-validation or a separate test set to get an unbiased estimate of the test set error. It is estimated internally, during the model building process.

Each tree is given a different bootstrap sample data set from the overall dataset. About one-third of the bootstrap are left out (Out of the Bag - OOB) and not used in the construction of the tree. This one-third of the bootstrap sample is then used for a test during the tree building process. At the end, the majority vote is used to determine what is the final classification of each of the observations from the outcomes of the Tests. The proportion of times that that each of the observation is not equal to the true class of averaged over all cases is the OOB error estimate.

Feature Importance

Feature importance refers to the mechanism of ability of the model to assign a score to the predictor features that indicates how important they are to the model in predicting the target.

Feature importance is important to the business to know where in the business to focus on to make improvements to achieve goals. E.g. It is important for a bank to know that age and education levels are large influencers of Credit Default. They would then factor these features in Modelling interest rates for certain risky age/education ranges.

The ‘feature_importances_’ attribute of the RandomForestClassifier model holds the data on Feature importances and we will see a visual representation in the practical section.

Random Forest in Practice

We will now see a Classification problem being solved using a Random Forest model. We have an input dataset containing Credit history of individuals with a class attribute name 'Defaulted' which contains one of two possible values – 1 (Defaulted) and 0 (not Defaulted). The input dataset contains two types of data – Demographic (Age, Sex, Marriage, etc.) and Behavioral data related to the Credit (past loans, payment, number of times a credit payment has been delayed by the customer etc.).

We will use a Random Forest Algorithm to train the model and then use it to predict on test data set aside from the Input dataset that we have. We will also run a Cross Validation with Hyperparameter Tuning to see what optimum values of the Hyperparameters produce the best results and how to arrive at them.

As the first step, we import the required libraries.

```
# Importing the required libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline

# Importing test_train_split from sklearn library
from sklearn.model_selection import train_test_split

# Importing random forest classifier from sklearn library
from sklearn.ensemble import RandomForestClassifier

# Importing classification report and confusion matrix from sklearn
metrics
from sklearn.metrics import classification_report,confusion_matrix,
accuracy_score

# To ignore warnings
import warnings
warnings.filterwarnings("ignore")
```

Import the Input data set present in CSV format into a Pandas DataFrame.

```
# Reading the csv file and putting it into 'df' object.
df = pd.read_csv('credit-card-default.csv')
df.head()
```

Output:

	ID	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_0	PAY_2	PAY_3	PAY_4	...	BILL_AMT4	BILL_AMT5	BILL_AMT6	PAY_AMT1	PAY_AMT2	PAY_A
0	1	20000	2	2	1	24	2	2	-1	-1	...	0	0	0	0	689	
1	2	120000	2	2	2	26	-1	2	0	0	...	3272	3455	3261	0	1000	
2	3	90000	2	2	2	34	0	0	0	0	...	14331	14948	15549	1518	1500	
3	4	50000	2	2	1	37	0	0	0	0	...	28314	28959	29547	2000	2019	
4	5	50000	1	2	1	57	-1	0	-1	0	...	20940	19146	19131	2000	36681	1

Display the type of columns.

```
# Let's understand the type of columns
df.info()
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30000 entries, 0 to 29999
Data columns (total 25 columns):
ID          30000 non-null int64
LIMIT_BAL   30000 non-null int64
SEX         30000 non-null int64
EDUCATION   30000 non-null int64
MARRIAGE    30000 non-null int64
AGE         30000 non-null int64
PAY_0       30000 non-null int64
PAY_2       30000 non-null int64
PAY_3       30000 non-null int64
PAY_4       30000 non-null int64
PAY_5       30000 non-null int64
PAY_6       30000 non-null int64
BILL_AMT1   30000 non-null int64
BILL_AMT2   30000 non-null int64
BILL_AMT3   30000 non-null int64
BILL_AMT4   30000 non-null int64
```

- There are 30000 observations (rows) in the dataset.
- The columns are all of int64 type.
- No missing values in the column.
- So, we can straightaway go ahead and build the model.

The 'ID' field is just a sequence field for the data rows. We drop it as it does not bear any influence on the model.

```
df = df.drop('ID',axis=1)
```

Split the dataset into Features and Target Class label (Dependent Variable) – X and y sets.

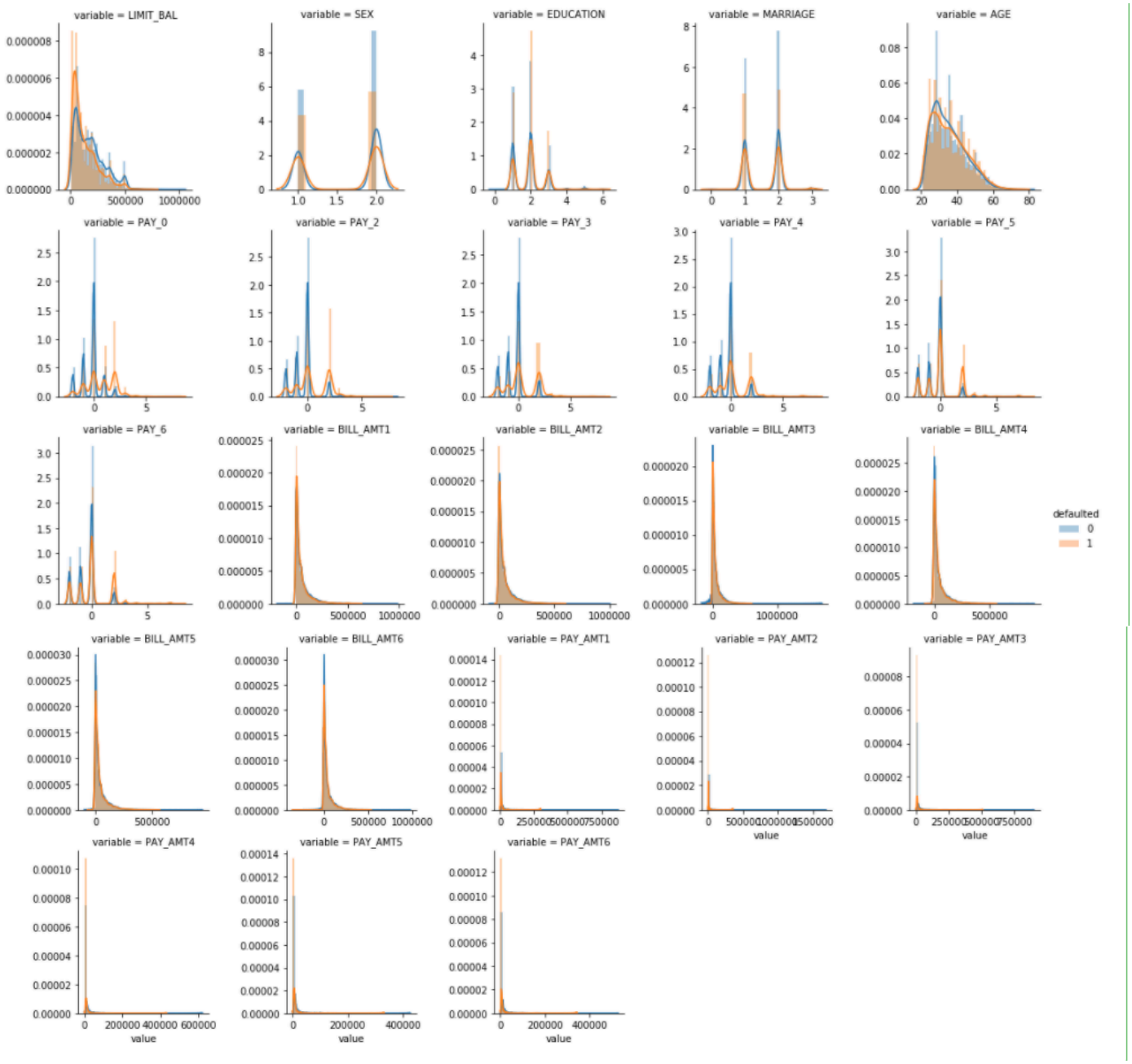
```
# Putting feature variable to X
X = df.drop('defaulted',axis=1)

# Putting response variable to y
y = df['defaulted']
```

Exploratory Data Analysis

We will now try to understand the relationships between the features and Defaulted / Non-Defaulted observations. We use the Seaborn library Distribution plot for this purpose.

```
output = 'defaulted'

cols = [ f for f in df.columns if df.dtypes[ f ] != "object"]
cols.remove( output )

f = pd.melt( df, id_vars=output, value_vars=cols)
g = sns.FacetGrid( f, hue=output, col="variable", col_wrap=5,
sharex=False, sharey=False )
g = g.map( sns.distplot, "value", kde=True).add_legend()
```

Output:



Some of the observations from the above distribution plots.

- People with a Lower LIMIT_BAL default more
- Higher proportion of Females (SEX=2) default
- More Educated people are defaulting more (wow!!)
- Slightly higher proportion of Singles compared to their Married counterparts Default
- Age range 20-40 have a higher propensity to default
- Non-Default observations have a MUCH higher affinity towards zero or negative PAY_X values.

You can carry out a bit more in-depth EDA. One example is given below to find out the distribution of Defaults/Non-Defaults in numeric terms.

```
print("We have %d defaults with EDUCATION=0" % len(df.loc[ df["EDUCATION"]==0]))
print("We have %d defaults with EDUCATION=1" % len(df.loc[ df["EDUCATION"]==1]))
print("We have %d defaults with EDUCATION=2" % len(df.loc[ df["EDUCATION"]==2]))
print("We have %d defaults with EDUCATION=3" % len(df.loc[ df["EDUCATION"]==3]))
print("We have %d defaults with EDUCATION=4" % len(df.loc[ df["EDUCATION"]==4]))
print("We have %d defaults with EDUCATION=5" % len(df.loc[ df["EDUCATION"]==5]))
print("We have %d defaults with EDUCATION=6" % len(df.loc[ df["EDUCATION"]==6]))
```

Output:

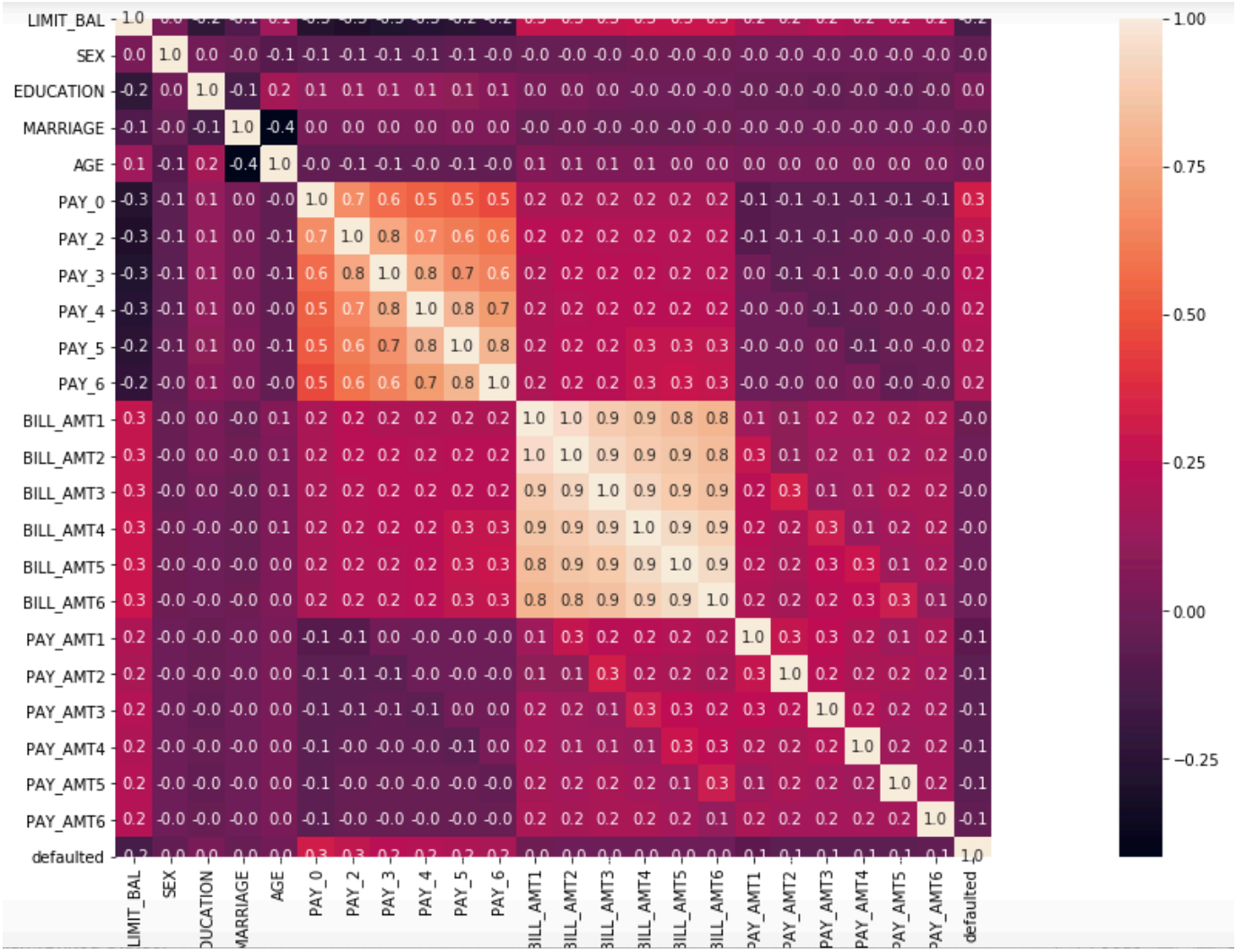
```
We have 14 defaults with EDUCATION=0
We have 10585 defaults with EDUCATION=1
We have 14030 defaults with EDUCATION=2
We have 4917 defaults with EDUCATION=3
We have 123 defaults with EDUCATION=4
We have 280 defaults with EDUCATION=5
We have 51 defaults with EDUCATION=6
```

The relationship between the class label and level of education is now more apparent from the above example. People with Education levels 1and 2 Default drastically more than others.

We will also find out the Correlation Matrix and Plot a Heatmap from the Corr values to find out about any relationships present between Feature (multicollinearity).

```
# Create a correlation Matrix and Heatmap
corr = df.corr()
plt.subplots(figsize=(30,10))
sns.heatmap( corr, square=True, annot=True, fmt=".1f" )
```

Output:



The lighter shaded blocks indicate higher degree of relationships. The Block of BILL_AMTx variables have multicollinearity between themselves and also there is little correlation with the 'defaulted' class label. We would drop these variables from the mix.

```
# Remove the BILL_AMT features
for i in range(1,7):
    X = X.drop("BILL_AMT" + str(i), axis=1)
print(X)
```

Output:

	LIMIT_BAL	SEX	EDUCATION	MARRIAGE	AGE	PAY_0	PAY_2	PAY_3	PAY_4	PAY_5	PAY_6	PAY_AMT1	PAY_AMT2	PAY_AMT3	PAY_AMT4	PAY_AM
0	20000	2	2	1	24	2	2	-1	-1	-2	-2	0	689	0	0	
1	120000	2	2	2	26	-1	2	0	0	0	2	0	1000	1000	1000	
2	90000	2	2	2	34	0	0	0	0	0	0	1518	1500	1000	1000	1(
3	50000	2	2	1	37	0	0	0	0	0	0	2000	2019	1200	1100	1(
4	50000	1	2	1	57	-1	0	-1	0	0	0	2000	36681	10000	9000	8
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
29995	220000	1	3	1	39	0	0	0	0	0	0	8500	20000	5003	3047	5(
29996	150000	1	3	2	43	-1	-1	-1	-1	0	0	1837	3526	8998	129	
29997	30000	1	2	2	37	4	3	2	-1	0	0	0	0	22000	4200	2(
29998	80000	1	3	1	41	1	-1	0	0	0	-1	85900	3409	1178	1926	52
29999	50000	1	2	1	46	0	0	0	0	0	0	2078	1800	1430	1000	1(

30000 rows × 17 columns

Develop the Classification Model

We will now split the X and y datasets into Train and Test sets. We will retain 70% of the data for Training the Random Forest model and remaining 30% of the data will be used for testing.

```
# Splitting the data into train and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.30, random_state=101)
```

The next step is to create the Random Forest Classifier object and fit the training data. This creates the model.

```
# Running the random forest with default parameters.
rfc = RandomForestClassifier()
rfc.fit(X_train,y_train)
```

Output:

```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=None, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=100,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

The above snapshot shows the default values of the Hyperparameters that are used by the model.

With this model, we will run a prediction on the Test data followed by checking out the stats on accuracy.

```
# Making predictions
predictions = rfc.predict(X_test)
print(classification_report(y_test,predictions))
print(confusion_matrix(y_test,predictions))
print(accuracy_score(y_test,predictions))
```

Output:

```
              precision    recall  f1-score   support

           0       0.84      0.95      0.89      7058
           1       0.65      0.36      0.46      1942

    accuracy                           0.82      9000
   macro avg       0.75      0.65      0.68      9000
weighted avg       0.80      0.82      0.80      9000

[[6677  381]
 [1241  701]]

0.8197777777777778
```

So, we have achieved an accuracy of 82% which is decent. Now, we will work with Hyperparameters to improve the model.

Before we do that, we should outline what these Hyperparameters are and how they help in improving the model.

- n_estimators: integer, optional: The number of trees in the forest. The higher the value of this parameter, the better generalised the model is. However, it also take a toll on the performance and do not improve the model after a certain point, hence, you should arrive at an optimal value.
- criterion: string, optional (default="gini"). This parameter determines the Homogeneity measurement mechanism for the Tree. Possible values are 'gini', 'entropy'.
- max_features: int, float, string or None, optional (default="auto"). Random Forest creates Trees with Random Sample of Features. This parameter determines the max number of Features in a Tree. Limiting this will limit the overlap of Features in various Tree, increase diversity and generalisability of the Tree.
- max_depth: integer or None, optional (default=None). The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples.
- min_samples_split : int, float, optional (default=2). The minimum number of samples required to split the current node.
    - If int, then consider min_samples_split as the minimum number of samples.
    - If float, then min_samples_split is a percentage.
- min_samples_leaf: int, float, optional (default=1)The minimum number of samples required to be at the resultant leaf node:
    - If int, then consider min_samples_leaf as the minimum number.
    - If float, then min_samples_leaf is a percentage.
- min_weight_fraction_leaf: float, optional (default=0). The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.
- max_leaf_nodes: int or None, optional (default=None). Grow trees with max_leaf_nodes in best-first fashion. Best nodes are defined as relative reduction in impurity. If None then unlimited number of leaf nodes.
- min_impurity_split: float. Threshold for early stopping in tree growth. A node will split if its impurity is above the threshold, otherwise it is a leaf.

Let's first try to find the optimum value of the 'max_depth' Hyperparameter. In the following code, we are using an iterative approach using GridSearchCV using a starting value of 2 and moving up to 10 with an increment of 1 each time.

```
# GridSearchCV to find optimal max_depth

from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV

# specify number of folds for k-fold CV
n_folds = 5

# parameters to build the model on
parameters = {'max_depth': range(2, 10, 1)}

# instantiate the model
rf = RandomForestClassifier()

# fit tree on training data
rf = GridSearchCV(rf, parameters,
                    cv=n_folds,
                   scoring="accuracy", return_train_score=True)
rf.fit(X_train, y_train)
```

Output:

```
GridSearchCV(cv=5, error_score=nan,
             estimator=RandomForestClassifier(bootstrap=True, ccp_alpha=0.0,
                                              class_weight=None,
                                              criterion='gini', max_depth=None,
                                              max_features='auto',
                                              max_leaf_nodes=None,
                                              max_samples=None,
                                              min_impurity_decrease=0.0,
                                              min_impurity_split=None,
                                              min_samples_leaf=1,
                                              min_samples_split=2,
                                              min_weight_fraction_leaf=0.0,
                                              n_estimators=100, n_jobs=None,
                                              oob_score=False,
                                              random_state=None, verbose=0,
                                              warm_start=False),
             iid='deprecated', n_jobs=None,
             param_grid={'max_depth': range(2, 10)}, pre_dispatch='2*n_jobs',
             refit=True, return_train_score=True, scoring='accuracy',
             verbose=0)
```

Note the default values of other Hyperparameters from the above output.

Let's print the scores from the Cross Validation.

```
# scores of GridSearch CV
scores = rf.cv_results_
pd.DataFrame(scores).head()
```

Output:

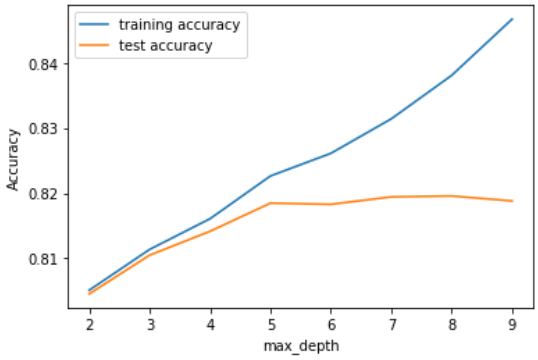
	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_max_depth	params	split0_test_score	split1_test_score	split2_test_score	split3_te
0	0.727647	0.047090	0.032294	0.003325	2	{'max_depth': 2}	0.810238	0.806905	0.802143	(
1	0.804564	0.013098	0.031219	0.000163	3	{'max_depth': 3}	0.811190	0.815000	0.811429	(
2	0.968274	0.018879	0.035305	0.003309	4	{'max_depth': 4}	0.816429	0.820238	0.812619	(
3	1.181660	0.051920	0.037056	0.002362	5	{'max_depth': 5}	0.816190	0.822857	0.820952	(
4	1.325248	0.074058	0.043022	0.006584	6	{'max_depth': 6}	0.815000	0.821905	0.819286	(

5 rows × 21 columns

We then plot the Accuracy scores from the Cross Validation for the range of values of max_depth we used. This will let us determine at what value of this Hyperparameter, the accuracy is highest and stable.

```
# plotting accuracies with max_depth

plt.figure()
plt.plot(scores["param_max_depth"],
         scores["mean_train_score"],
         label="training accuracy")
plt.plot(scores["param_max_depth"],
         scores["mean_test_score"],
         label="test accuracy")
plt.xlabel("max_depth")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```

Output:



The optimum value of 'max_depth' comes to be 5 from this experiment. After that, it stabilizes.

The ‘n_estimator’ parameter tells the model as to how many trees have to be created in the Random Forest. The higher the number of trees is better for the model generalizability. The trade-off with performance and the value at which it flattens need to be established. We use a range of 100 to 1000 with an increment of 150 each time the Cross Validation iterates.

```
# GridSearchCV to find optimal n_estimators
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV

# specify number of folds for k-fold CV
n_folds = 5

# parameters to build the model on
parameters = {'n_estimators': range(100, 1000, 150)}

# instantiate the model (note we are specifying a max_depth)
rf = RandomForestClassifier(max_depth=5)

# fit tree on training data
rf = GridSearchCV(rf, parameters,
                    cv=n_folds,
                   scoring="accuracy", return_train_score=True)
rf.fit(X_train, y_train)
```

Output:

```
GridSearchCV(cv=5, error_score=nan,
             estimator=RandomForestClassifier(bootstrap=True, ccp_alpha=0.0,
                                              class_weight=None,
                                              criterion='gini', max_depth=5,
                                              max_features='auto',
                                              max_leaf_nodes=None,
                                              max_samples=None,
                                              min_impurity_decrease=0.0,
                                              min_impurity_split=None,
                                              min_samples_leaf=1,
                                              min_samples_split=2,
                                              min_weight_fraction_leaf=0.0,
                                              n_estimators=100, n_jobs=None,
                                              oob_score=False,
                                              random_state=None, verbose=0,
                                              warm_start=False),
             iid='deprecated', n_jobs=None,
             param_grid={'n_estimators': range(100, 1000, 150)},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
             scoring='accuracy', verbose=0)
```

Printing the scores from Cross Validation Iterations.

```
# scores of GridSearch CV
scores = rf.cv_results_
pd.DataFrame(scores).head()
```

Output:

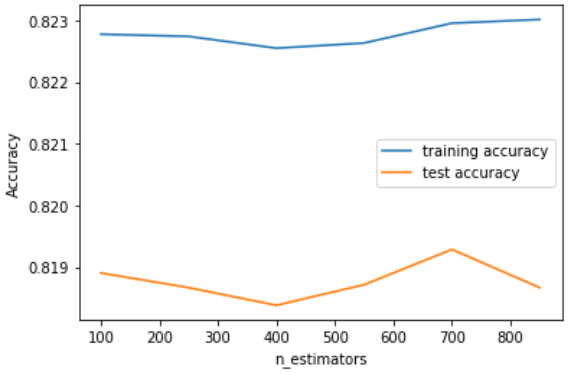
	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_n_estimators	params	split0_test_score	split1_test_score	split2_test_score	split3
0	1.163330	0.071986	0.037749	0.003818	100	{'n_estimators': 100}	0.817857	0.821429	0.822381	
1	2.759818	0.050243	0.085715	0.002523	250	{'n_estimators': 250}	0.817619	0.820714	0.820476	
2	4.456365	0.082712	0.143901	0.008309	400	{'n_estimators': 400}	0.817381	0.822381	0.819524	
3	6.076428	0.051208	0.184474	0.003766	550	{'n_estimators': 550}	0.817143	0.822619	0.821190	
4	7.773267	0.078442	0.273052	0.057604	700	{'n_estimators': 700}	0.817381	0.822143	0.820952	

5 rows × 21 columns

We then plot the Accuracy scores from the Cross Validation for the range of values of 'n_estimators' we used. This will let us determine at what value of this Hyperparameter, the accuracy is highest and stable.

```
# plotting accuracies with n_estimators
plt.figure()
plt.plot(scores["param_n_estimators"],
         scores["mean_train_score"],
         label="training accuracy")
plt.plot(scores["param_n_estimators"],
         scores["mean_test_score"],
         label="test accuracy")
plt.xlabel("n_estimators")
plt.ylabel("Accuracy")
plt.legend()
plt.show()
```

Output:



The optimum value of 'n_estimators' comes to be 700 from this experiment before it starts going down.

We can run our Cross Validation a bit smartly by creating a parameter grid with all the Hyperparameters we want to test and finding the best combination at one go.

```
# Create the parameter grid based on the results of random search
param_grid = {
    'max_depth': [3,8,1],
    'min_samples_leaf': range(100, 400, 100),
    'min_samples_split': range(200, 500, 100),
    'n_estimators': [100,1000, 150],
    'max_features': [5, 10, 1]
}

# Create a based model
rf = RandomForestClassifier()

# Instantiate the grid search model
grid_search = GridSearchCV(estimator = rf, param_grid = param_grid,
                           cv = 3, n_jobs = -1,verbose = 1,
return_train_score=True)

grid_search.fit(X_train, y_train)
```

Output:

```
Fitting 3 folds for each of 243 candidates, totalling 729 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  42 tasks      | elapsed:  1.4min
[Parallel(n_jobs=-1)]: Done 192 tasks      | elapsed:  6.8min
[Parallel(n_jobs=-1)]: Done 442 tasks      | elapsed: 19.4min
[Parallel(n_jobs=-1)]: Done 729 out of 729 | elapsed: 25.2min finished
GridSearchCV(cv=3, error_score=nan,
             estimator=RandomForestClassifier(bootstrap=True, ccp_alpha=0.0,
                                              class_weight=None,
                                              criterion='gini', max_depth=None,
                                              max_features='auto',
                                              max_leaf_nodes=None,
                                              max_samples=None,
                                              min_impurity_decrease=0.0,
                                              min_impurity_split=None,
                                              min_samples_leaf=1,
                                              min_samples_split=2,
                                              min_weight_fraction_leaf=0.0,
                                              n_estimators=100, n_jobs=None,
                                              oob_score=False,
                                              random_state=None, verbose=0,
                                              warm_start=False),
             iid='deprecated', n_jobs=-1,
             param_grid={'max_depth': [3, 8, 1], 'max_features': [5, 10, 1],
                         'min_samples_leaf': range(100, 400, 100),
                         'min_samples_split': range(200, 500, 100),
                         'n_estimators': [100, 1000, 150]},
             pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
             scoring=None, verbose=1)
```

We extract the best combination from the gri_search Cross Validation object.

```
# printing the optimal accuracy score and hyperparameters
print('We can get accuracy
of',grid_search.best_score_,'using',grid_search.best_params_)
```

Output:

```
We can get accuracy of 0.8185238095238095 using {'max_depth': 8, 'max_features': 10, 'min_samples_leaf': 100, 'min_sa
mples_split': 200, 'n_estimators': 150}
```

And then we use the best parameter values and run the Random Forest learning again to get the final model.

```
# model with the best hyperparameters
from sklearn.ensemble import RandomForestClassifier
rfc = RandomForestClassifier(bootstrap=True,
                             max_depth=8,
                             min_samples_leaf=100,
                             min_samples_split=200,
                             max_features=10,
                             n_estimators=150)

# fit
rfc.fit(X_train,y_train)
```

Output:

```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=8, max_features=10,
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=100, min_samples_split=200,
                       min_weight_fraction_leaf=0.0, n_estimators=150,
                       n_jobs=None, oob_score=False, random_state=None,
                       verbose=0, warm_start=False)
```

And finally, we predict on the Test dataset and display the accuracy stats.

```
# predict
predictions = rfc.predict(X_test)

print(classification_report(y_test,predictions))

print(confusion_matrix(y_test,predictions))
```

Output:

```
              precision    recall  f1-score   support

           0       0.84      0.96      0.90      7058
           1       0.69      0.36      0.47      1942

    accuracy                           0.83      9000
   macro avg       0.77      0.66      0.68      9000
weighted avg       0.81      0.83      0.80      9000
```

```
[[6752  306]
 [1250  692]]
```

We have achieved an accuracy of nearly 83% on the test data which is same as Train score which states the good generalizability we have achieved.

Feature Importance

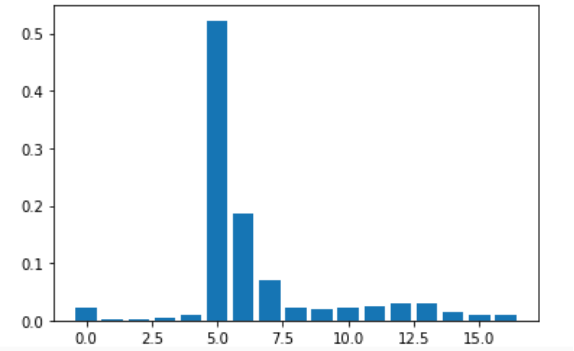
One of the important advantages of Random Forest is that we can extract Feature Importance from the model. In the below code, we are displaying how individual features influence the final model.

```
# Printing Features Importances

from matplotlib import pyplot
importance = rfc.feature_importances_
for i, v in enumerate(importance):
    print('Feature: %0d, Score: %.5f' % (i,v))

# plot feature importance
pyplot.bar([x for x in range(len(importance))], importance)
pyplot.show()
```

Output:

```
Feature: 0, Score: 0.02255
Feature: 1, Score: 0.00175
Feature: 2, Score: 0.00176
Feature: 3, Score: 0.00344
Feature: 4, Score: 0.00915
Feature: 5, Score: 0.52232
Feature: 6, Score: 0.18531
Feature: 7, Score: 0.07043
Feature: 8, Score: 0.02319
Feature: 9, Score: 0.02035
Feature: 10, Score: 0.02117
Feature: 11, Score: 0.02489
Feature: 12, Score: 0.02854
Feature: 13, Score: 0.03086
Feature: 14, Score: 0.01474
Feature: 15, Score: 0.00898
Feature: 16, Score: 0.01059
```

In the output here, we have the individual Feature scores, that's a relative score that helps determine comparatively, which features have the most influence on the target classification through the model.



This is a bar graph representation of the same information above on Feature scores. Apparently the 5th attribute has the most influence on the model.

Explore Teksands.ai
Learn from Researchers from IITs/NITs

Follow us in Social Media

https://www.facebook.com/Teksands

https://www.linkedin.com/company/teksands/

https://twitter.com/teksands

https://www.instagram.com/teksands.ai/

Call or WhatsApp us to Enquire about our Machine Learning/AI and Data Science Courses

https://api.whatsapp.com/send?phone=916362732428