5 DECISION TREES, RANDOM FOREST, AND MULTI-CLASS PREDICTION

Nonparametric estimation is a statistical method that estimates the functional form of data fits when parametric modelling is absent and where there is no guidance from theory. Examples of nonparametric estimation are kNN, SVM, and kernel estimation method akin to fitting histograms on frequency counts to get a distributional form.

Decision Tree (DT) is a method to predict the value of a target variable (whether predicting a class/category or a continuous value) by a non-parametric way without employing some parametric functional form relating target to features such as in a logistic regression approach. A DT contains decision rules that are conditions to test each sample point or instance on the training data set. For example, to partition red strawberries from red rambutans (an Australian friend of mine years ago called the latter 'hairy strawberries'), one could use the decision rule that if the fruit has thick hairs, then it is rambutan; otherwise, it is strawberry. Another example could be trying to partition between crocodiles and alligators. A decision rule could be that if the creature has rounded snout, then it is an alligator; otherwise, it is a crocodile. A difficult example is identifying a mugger from a police lineup where the other people in the lineup are innocent. One could use a decision rule based on observed features of the lineup suspects, such as whether the suspect has a proven alibi at the time of the crime. This may reduce the number of suspects, but there could still be many who are left, so more decision rules must be added sequentially to filter out the innocent ones.

The DT approach is typically supervised learning with a training data set, a validation data set and a test data set. In this chapter, we look at the usefulness of DT and the related Random Forest (RF) method of predicting binary classification. It is called the Categorical Variable DT when predicting binary or categorical groups. When used for predicting continuous value outputs, the DT may be called a Continuous Variable DT. Categorical Variable DT (and related RF) is generally found to be more accurate relative to use of Continuous Variable DT as the latter involves many more permutations of the target variable. However, due to the non-parametric nature, DT methods can produce unstable outcomes when data features have high variabilities as the

decision rule outcomes on the partitioning can change sharply from one test data point to another.

5.1 Decision Tree Method

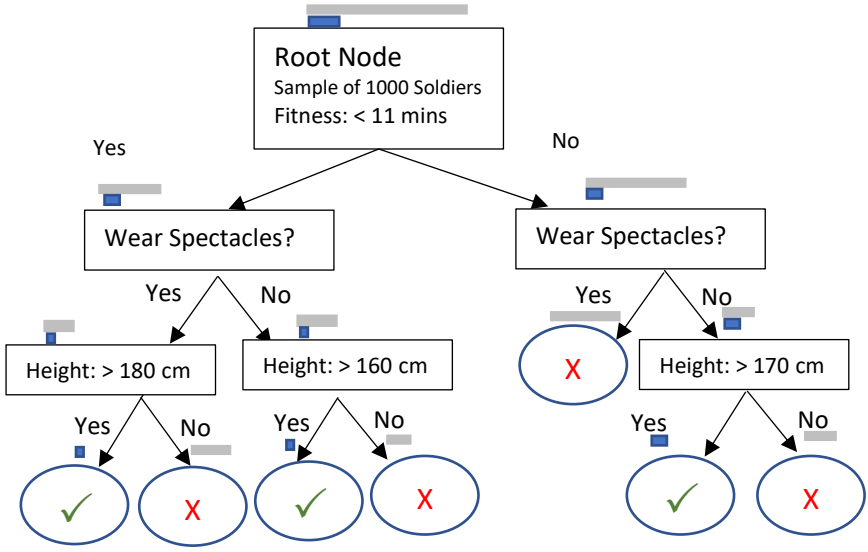
Suppose we use DT to predict if a case belongs to the positive or the negative class, being binary partitions of all cases or instances, i.e., any case could be put into the positive or the negative class, and that the classes are reasonably balanced. Balance refers to the number of cases in the positive class being of similar magnitude to the number of cases in the negative class. Let each case, besides its positive or negative label, also carry a set of attributes or features that can help in predicting its type/label.

We illustrate as follows how a DT can help to predict (1) if a soldier who passed BMT is a marksman (defined as one who shoots on target at least 84% of the times or 21 hits out of 25 shots at training targets), with only preliminary training, given the soldier's fitness (time in minutes to complete a 2.4 km run), height (in cm) and whether the soldier needs to wear spectacles or not; and (2) if a person would get a certain lung disease by age 60 given the person's fitness measured by number of hours of exercise a week, whether a regular smoker (at least one cigarette a day) or not, and whether there is a hereditary factor, i.e., if the person's parent (one or both) has had such a disease.

A trained DT based on a training sample of past 1000 soldiers for (1) could be as follows. The first root node is also a decision node on using fitness to split the total sample of 1000 soldiers. Value of a node refers to the number of positive cases (training sample marksmen) and number of negative cases (non-marksmen). For the illustration, this set of values is indicated by the relative length of the dark shade versus lighter grey bars above the decision node. The decision node is a node whereby a decision is made, e.g., if a sample point or case or instance (a soldier in the training data set) has fitness-run below 11 minutes (or else equal and above).

A feature is chosen (in some optimal way to be explained later) and a threshold of the feature is chosen. The latter refers to some values of the feature as the splitting point to divide the data associated with the feature into two subsets. The splitting or branching is based on 'Yes' or 'No' to the outcome of the decision rule. If the feature is categorical, then the splitting is based on 'Yes' or 'No' to the categorical rule. The decision node is associated with a decision rule. Data is split into subsets and moves along the branches to the

next (lower) levels of decision nodes until the leaf or terminal nodes are reached. At each leaf (end) node, the model makes a prediction.



The rectangular boxes represent the decision nodes where splitting of the sample then occurs along the subsequent branches. The ovals at the end of the tree (upside down) represents the leaf nodes or terminal nodes. These carry the prediction or final decision whether prediction (tick) is a marksman or (cross) not a marksman. In this example, at each terminal node or leaf, the remaining sub-sample is either all marksman or not marksmen. Hence there is no more sub-dividing. The decision tree could also end in a leaf with no further extension or decision box if no other features can be of help in further separation of the types in the leaf.

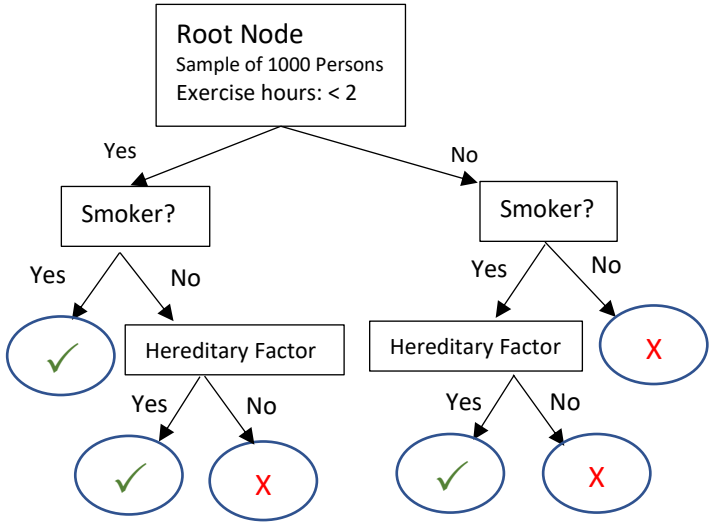
Recall that the bars above the boxes or ovals represent the number of marksmen (dark shade) and non-marksmen (lighter shade) from the historical sample used as training set. With each decision and split, the numbers of marksmen and non-marksmen are split into the next set of nodes.

The prediction rules according to this DT are: if fitness is < 11 mins, no spectacles, and height > 160 cm, or if fitness is < 11 mins, spectacles, and height > 180 cm, or if fitness is ≥ 11 mins, no spectacles, and height > 170 cm, then marksman; otherwise not in the other feature conditions.

In general, it is seen that with a given training set, one could continue to search for new features (associated with the targets) such that a chain of rules would lead to perfect discrimination – separating the targets into the different categories. This could be over-fitting and may not produce a good accuracy on the test set. Some regularizations/constraints on the hyperparameters may be introduced such as limiting the depth (levels) of the DT instead of the default maximum (8 to 32 in the sklearn app), limiting to no decision node based on a feature if the improvement in the differentiation is minimal, limiting the total number of nodes, limiting to a minimum number of sample points or cases on any node and not to split further, and so on.

Another example is a DT based on a training sample of past 1000 medical insurance buyers over a long period. Assume no moral hazard problem and that each insurance buyer starts off under the insurance scheme with no sign of contracting the disease.

A tick at the leaf denotes prediction of getting the disease; otherwise, it is a cross. The prediction rules according to this DT are: if exercise hours is < 2 per week, if smoker, or, if exercise hours is < 2 per week, if non-smoker but with hereditary factor, then disease is predicted. Or else, if exercise hours is ≥ 2 per week, smoker and with hereditary factor, then disease is predicted. Otherwise, disease is not predicted.



Next, we explain how the DT chooses which feature to create the split, how is the split condition chosen, and when does the splitting stop. We continue with the marksman example.

At the root node, the sample (size N) provides a measure of Gini impurity that is defined as $1 - \sum_{k=1}^{m} p_k^2$ where m is the number of different categories in the sample and p_k is the empirical probability of being in category k. Thus, in the marksmen case, if the original sample has 200 marksmen and 800 non-marksmen, then if the positive case 1 (marksmen) has probability p_1 = 200/1000 = 0.2, and the negative case 2 (non-marksmen) has probability p_2 = 0.8. The Gini impurity would be $1 - (0.2^2 + 0.8^2) = 0.32$ in this binary classification. Note that if $p_1 = p_2 = 0.5$, then we have a maximum Gini impurity. But $p_1=p_2=0.5$ is also the case for maximum entropy (uncertainty in distributional outcome) in the log loss or (cross-) entropy measure of $-\sum_{k=1}^{m} p_k \ln(p_k)$. Hence lower Gini impurity is related to lower entropy or better resolution of the classification rules. The idea is to develop rules leading to the lowest Gini impurity or else lowest entropy in the next nodes. For binary classifications, m = 2, so depending on the optimization criterion, we either minimize $1 - \sum_{k=1}^{2} p_k^2$ or else the cross-entropy measure $-\sum_{k=1}^{2} p_k \ln(p_k)$. Both these measures are close approximations of each other. Computing Gini impurity is more efficient as it does not involve finding logarithms in the cross-entropy measure.

The minimization of loss function as Gini impurity is done by (1) working on each feature considering splitting at different points – for a continuous variable feature, it means discretizing it and considering splits at each discrete points, e.g., splits at < 9 mins, < 9.1 mins, < 9.2 mins, ….., < 11 mins, < 11.1 mins, ….., <14 mins etc., and then (2) computing the Gini impurities in the new next layer binary nodes under each split, and (3) finding that split that produces the lowest weighted Gini impurities lower than that in the previous node (hence the largest gain in information – largest drop in entropy in the new split). For discrete valued features – the potential split point could be the mid-points of adjacent discrete values of the feature.

For example, the following could be a result of the split. The numbers in the brackets ( ) denote number of marksmen on the left and number of non-marksmen on the right. There are a total of 200 marksmen and 800 non-marksmen in the training data set. Consider the split at Fitness < 10.9 minutes. The Gini impurity for the 'Yes' branch is $1 - (100/420)^2 - (320/420)^2 =$

0.36281. The Gini impurity for the 'No' branch is $1 - (100/580)^2 - (480/580)^2 = 0.28358$. Then the weighted average of Gini impurities in this next layer of binary nodes under the split is $(420/1000) \times 0.36281 + (580/1000) \times 0.28358 = 0.31790$.

(Values) [Gini Impurity]	Yes	No	Weighted Gini Impurity
If < 10.9 min	(100,320) [0.36281]	(100,480) [0.28358]	$(420/1000) \times 0.36281 + (580/1000) \times 0.28358 = 0.31790$
If < 11 min	(120,300) $[1 - (120/420)^2 - (300/420)^2 = 0.40816]$	(80,500) $[1 - (80/580)^2 - (500/580)^2 = 0.23781]$	$(420/1000) \times 0.40816 + (580/1000) \times 0.23781 = 0.30936$
If < 11.1 min	(110,340) [0.369383]	(90, 460) [0.273719]	$(450/1000) \times 0.369383 + (550/1000) \times 0.273719 = 0.31677$

Suppose the split at Fitness < 11.0 min gives the lowest weighted Gini impurity or biggest drop in impurity from the previous root node of 0.32. Suppose also that considering splits in all the other features do not produce any bigger drop in impurity. Then the first decision node is whether Fitness < 11.0 min. Note that the Gini impurity is the loss function to be minimized in this DT.

Building the DT then continues to the next level (down) when the next best information gain feature would be used to create the next split, and so on. In case a leaf (final node) ends without fully discriminating against the two (or more) categories due to regularization (non-overfitting), the empirical probability in the sample in that leaf can be used to make the prediction. The splitting of the sample into smaller subsets along each branch is called recursive distribution of the sample. In a binary classification, if a terminal leaf has H number of positive cases and G number of negative cases, then the (empirical) probability of a positive case in the prediction would be H/(H+G) and that of a negative case would be G/(H+G).

Once the DT is trained using the training set, the hyperparameters are fine-tuned in a validation data set for optimal prediction. Then the DT could be re-

trained using a re-grouped training set (including validation data). We assume the validation has been done, and we proceeded with the training as seen earlier and then it is used next to predict the outcomes/categorizations in the test data set. All the test data X-variables (features) are used to decide on a positive or a negative case based on the trained decision rules. In the testing, the test sample point labels/types are not used for the prediction but are used afterward to measure the prediction performance.

We briefly explain the Continuous Variable DT as follows. Suppose in the same marksmen problem above, we want to predict the actual shooting score – how many shots on target out of a total of 25 shots in the shooting range test. This is a discrete variable – but any continuous variable in the finite initial sample is considered a discrete value (no doubt with decimal places in the numbers).

Continuous Variable DT works slightly differently whereby the loss criterion is squared error (default case) -- using DecisionTreeRegressor instead of DecisionTreeClassifier (for classification) in sklearn. Continuous Variable DT is sometimes called a regression tree model. Suppose we plot the actual shoot score of the 1000 cases/instance against one of the features – Fitness.

The diagram serves only to illustrate as there would be 1000 points or instances. Suppose at Fitness < 11.0 min, we obtain two partitions. On the left we have mean square error

$$\text{mse}_1 = \frac{\sum_{j=1}^{L}\left(Y_j - \bar{Y}^L\right)^2}{L}$$

where L is the number of sample points with feature Fitness < 11.0 min and on the right

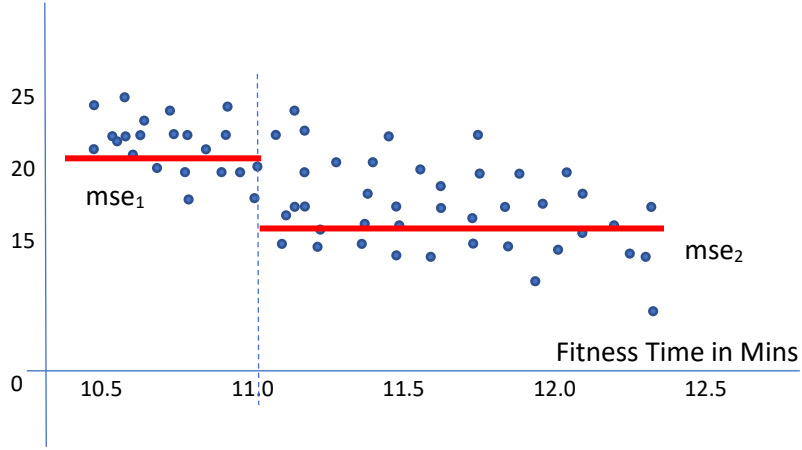
$$\text{mse}_2 = \frac{\sum_{j=1}^{R}\left(Y_j - \bar{Y}^R\right)^2}{R}$$

where R is the number of sample points with feature Fitness ≥ 11.0 min. Total L + R = 1000 from the total number of sample points/instances before the split from the earlier node. $\bar{Y}^L$ and $\bar{Y}^R$ are respectively the left sub-sample and right sub-sample means. Weighted variance (mean squared error) or loss function in this case is L/(L+R) × mse_1 + R/(L+R) × mse_2. This is the same as

$$[\ \sum_{j=1}^{L}\left(Y_j - \bar{Y}^L\right)^2 + \ \sum_{j=1}^{R}\left(Y_j - \bar{Y}^R\right)^2\ ]\ /(\text{L+R}).$$

The decision rule to split, i.e., at Fitness < 11 min, is chosen such that this loss function is the minimum across other possible splits and across all other

feature splits and is also a variance reduction from the previous higher-level node.



The next level split is then done on the left sub-sample, and another next level split is done separately on the right sub-sample using the next feature, and so on. The potential split points in the features are typically obtained as the mid-points of two adjacent discrete values of observations of the feature under consideration.

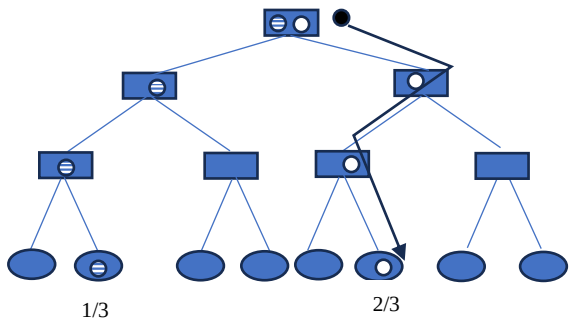
Finally, when all levels are done and the leaf node is on instances with the same shooting score, the DT can then be used to perform prediction of a future soldier's shooting score based on a new soldier's features. If the leaf ends with a small set of different shooting scores – then the mean can be taken as the predicted score given the decision rules along that DT ending in that leaf.

The method of splitting a feature via threshold to form a decision rule on a DT by minimizing a loss criterion is also called a classification and regression tree (CART) where more generally, the nodes carry not just the value information but also estimated values and derivatives of loss function computed at nodal level.

Just as in a categorical variable DT, once the DT is trained using the training set, it is then used to predict the outcomes in the test data set. All the test data X-variables (features) are used to decide on the predicted shooting score based on the trained decision rules. In the testing, the test sample point labels/types are not used for the prediction but are used afterward to measure the prediction performance.

The prediction in testing is illustrated as follows. The following is a trained decision tree showing the root node on top down to the leaf nodes. The DT and decision rules were arrived at during the training by making use of the knowledge of the training set labels.

A trained sample point from the training data set may be the white circle. Given its features and the decision rules on the features, it ended up in leaf node indicating 2/3 (if it is a categorical variable DT, i.e., label is categorical). The trained DT indicates that the leaf (terminal node) with 2/3 meant there were 2/3 positive cases and 1/3 negative case there in the training set. Essentially all the original training data set instances were distributed into the (terminal) leaves of the trained DT. Another training sample point could be the striped circle ending up in leaf node indicating 1/3. In this case, this test instance is predicted to be a positive case with only 1/3 probability, but a negative case with 2/3 probability.



Suppose the black circle is a test sample point or instance, and the decision rules in the fitted or trained tree led it to land on the leaf with 2/3 indicating it has a probability of 2/3 being a positive case. In this case of predicted probability, since the usual threshold is 0.5, predicted Prob (label Y=1) = 2/3 implies it is predicted as category Y=1.

If it is a continuous variable DT, then instances would end in a leaf with a numerical value, e.g., 12. This would mean the predicted target value is 12.

Note that for complex data, it is sometimes possible for the algorithm to use the same feature (used earlier for splitting closer to the root node) to split again at some points along a sub-tree. The way the splitting is done is based on the "greedy" algorithm approach that looks for the best option at hand without looking down the road if other not immediately best steps could

instead be taken that may actually produce overall lower Gini impurities in all leaves.

DTs and their ensemble seen later are not as sensitive to multi-collinearity in features as in other methods such as linear models or logistic regression. This is because for any highly collinear features, the DT proceeds by choosing only one to perform the decision rule. The other one does not at the same time affect the outcome of the DT.

DT can also be used to predict multi-class problems, e.g., predicting if a test sample point belongs to class A, class B, or class C. The approach is the same, except now each node contains value function with 3 numbers, viz., number of case A, number of case B, and number of case C. At each decision split into the next level of two nodes, the lowest weighting of Gini impurity $1 - \sum_{k=1}^{m} p_k^2$ is selected. The downward splitting of nodes continues until the leaf or end node. At each leaf, if Gini impurity is not zero (i.e., zero implies perfect identification of A or B or C in the training data), empirical probabilities of the cases are used and the class with the highest empirical probability (i.e., having the highest number of sample points belonging to that class in that leaf) is the predicted class.

5.2 Worked Example DT – Data

This data set corporate_rating2.csv is obtained on public website from Kaggle https://www.kaggle.com/ datasets/agewerc/corporate-credit-rating.

The dataset contains 2029 credit ratings issued by major agencies from 2010 to 2016 on large US companies. For each credit rating, each company shows 25 accounting features. The other 6 columns in the data set consist of rating, name of firm, symbol of firm in exchange trading, the rating agency, date of rating, and the sector of the firm. I added a new column called ‘Class” that contains “1” if the credit rating indicates investment grade, i.e., “AAA”, “AA”, “A”, or “BBB”. Any rating below “BBB” – considered as speculate grade – has a class value of “0”. There are 1165 instances of Class “1” and 864 instances of Class “0”.

The idea in this exercise using Chapter5-1.ipynb is to use the company financial/accounting variables/ratios to try to predict which binary class of investment grade or speculative grade the firm belongs to, using the published class as target for supervised training. A particular company may have several

data points in the sample set as it might have obtained credit ratings, hence a classification as “1” or “0”, at different calendar dates/times.

When adequately trained, the predictive model can be used to help new firms getting listed as well as investors/creditors of such firms to use the firm’s internal accounting ratios (reported as accurately as possible in a format close to the published ones) to help predict which category the firm should belong to. Obviously if identified as investment grade, the firm would find it easier to float new shares (IPOs) and issue debts at a cheaper interest cost.

In codeline [4], the features information is shown.

```
[4]: # Display the structure
     df_ratios.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2029 entries, 0 to 2028
Data columns (total 32 columns):
 #   Column                                Non-Null Count  Dtype
---  ------                                --------------  -----
 0   Rating                                2029 non-null   object
 1   Class                                 2029 non-null   int64
 2   Name                                  2029 non-null   object
 3   Symbol                                2029 non-null   object
 4   Rating Agency Name                    2029 non-null   object
 5   Date                                  2029 non-null   object
 6   Sector                                2029 non-null   object
 7   currentRatio                          2029 non-null   float64
 8   quickRatio                            2029 non-null   float64
 9   cashRatio                             2029 non-null   float64
 10  daysOfSalesOutstanding                2029 non-null   float64
 11  netProfitMargin                       2029 non-null   float64
 12  pretaxProfitMargin                    2029 non-null   float64
 13  grossProfitMargin                     2029 non-null   float64
 14  operatingProfitMargin                 2029 non-null   float64
 15  returnOnAssets                        2029 non-null   float64
 16  returnOnCapitalEmployed               2029 non-null   float64
 17  returnOnEquity                        2029 non-null   float64
 18  assetTurnover                         2029 non-null   float64
 19  fixedAssetTurnover                    2029 non-null   float64
 20  debtEquityRatio                       2029 non-null   float64
 21  debtRatio                             2029 non-null   float64
 22  effectiveTaxRate                      2029 non-null   float64
 23  freeCashFlowOperatingCashFlowRatio    2029 non-null   float64
 24  freeCashFlowPerShare                  2029 non-null   float64
 25  cashPerShare                          2029 non-null   float64
 26  companyEquityMultiplier               2029 non-null   float64
 27  ebitPerRevenue                        2029 non-null   float64
 28  enterpriseValueMultiple               2029 non-null   float64
 29  operatingCashFlowPerShare             2029 non-null   float64
 30  operatingCashFlowSalesRatio           2029 non-null   float64
 31  payablesTurnover                      2029 non-null   float64
```

Standard scaling is done in code line [8] on the features, excepting the Class target variable. The scaling reduces the variances of the features.

[8]:
```
# Normalization
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler().fit(df_ratios_1)
df_ratios_1 = scaler.transform(df_ratios_1)
df_ratios_1=pd.DataFrame(df_ratios_1,\
     columns=['currentRatio','quickRatio','cashRatio','daysOfSalesOutstanding',\
     'netProfitMargin', 'pretaxProfitMargin', 'grossProfitMargin',\
     'operatingProfitMargin','returnOnAssets','returnOnCapital',\
     'returnOnEquity','assetTurnover','fixedAssetTurnover','debtEquityRatio',\
     'debtRatio','effectiveTaxRate','freeCashFlowOperatingCashFlowRatio',\
      'freeCashFlowPerShare','cashPerShare','companyEquityMultiplier',\
      'ebitPerRevenue','enterpriseValueMultiple','operatingCashFlowPerShare',\
      'operatingCashFlowSalesRatio','payablesTurnover'])
```

This scaled data is then combined/concatenated with the Class variable in [13]. Note that we do not scale the Class variable here.

[13]:
```
TT=pd.concat([df_ratios['Class'],df_ratios_new],axis=1)
TT.shape
TT.tail()
```

[13]:

	Class	currentRatio	quickRatio	cashRatio	daysOfSalesOutstanding	netProfitMargin	...
2024	1	0.186827	0.135444	1.783407	-0.074822	-0.036575	...
2025	0	-0.012870	-0.041268	-0.066739	-0.068183	-0.073966	...
2026	0	-0.060074	-0.054997	-0.120995	-0.041872	-0.047159	...
2027	0	-0.059442	-0.057857	-0.099558	-0.045460	-0.031518	...
2028	0	-0.055507	-0.049416	-0.129463	-0.040959	-0.084191	...

5 rows × 26 columns

The total sample is then randomly split into 75% training set (1521 sample points) and 25% test set (508 sample points).

[15]:
```
Train, Test = train_test_split(TT,
                               test_size=0.25,
                               random_state=0)

X_train, y_train = Train.iloc[:,1:26], Train.iloc[:,0]
### choose the 25 features
X_test, y_test = Test.iloc[:,1:26], Test.iloc[:,0]
```

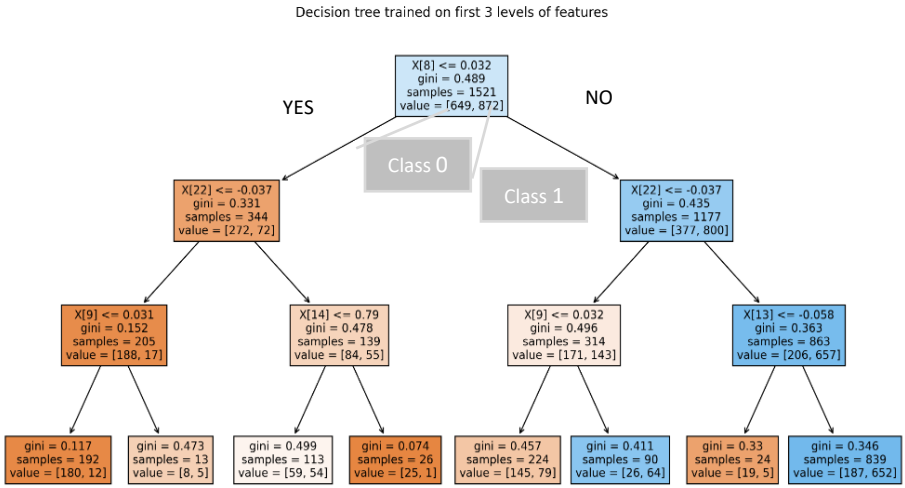
[16]:
```
X_train.shape, y_train.shape, X_test.shape, y_test.shape
```

[16]:
```
((1521, 25), (1521,), (508, 25), (508,))
```

The sklearn DecisionClassifier algorithm is then performed on the training data set. See codeline [19]. In this preliminary DT classification, only a depth of 3 levels is applied. The root node and decision nodes and the branching are shown.

Decision Tree

[19]:
```
from matplotlib import pyplot as plt
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import plot_tree
### The following is experimental DT with only depth of 3 levels
dt1 = DecisionTreeClassifier(max_depth=3,criterion='gini').fit(X_train, y_train)

fig, axes = plt.subplots(nrows = 1,ncols = 1,figsize = (15,8), dpi=300)
plot_tree(dt1, filled=True)
plt.title("Decision tree trained on first 3 levels of features")
plt.show()
```



The text version of the tree is shown as follows.

[20]:
```
from sklearn.tree import export_text
text_representation = export_text(dt1)
print(text_representation)
```

```
|--- feature_8 <= 0.03
|   |--- feature_22 <= -0.04
|   |   |--- feature_9 <= 0.03
|   |   |   |--- class: 0
|   |   |--- feature_9 >  0.03
|   |   |   |--- class: 0
|   |--- feature_22 >  -0.04
|   |   |--- feature_14 <= 0.79
|   |   |   |--- class: 0
|   |   |--- feature_14 >  0.79
|   |   |   |--- class: 0
|--- feature_8 >  0.03
|   |--- feature_22 <= -0.04
|   |   |--- feature_9 <= 0.03
|   |   |   |--- class: 0
|   |   |--- feature_9 >  0.03
|   |   |   |--- class: 1
|   |--- feature_22 >  -0.04
|   |   |--- feature_13 <= -0.06
|   |   |   |--- class: 0
|   |   |--- feature_13 >  -0.06
|   |   |   |--- class: 1
```

There are 649 instances of Class "0" out of 1521 in the training sample or a fraction of 0.4267. Hence the Gini impurity at root node is computed as $1–0.4267^2–0.5733^2 = 0.489$. The first decision rule at the root node is found in the feature 8 – Operating Profit Margin – a key accounting variable of a firm. The rule based on the scaled number is that if its value < 0.032, then go to the left branch, if not go to the right branch.

The Gini impurity calculated based on the left branch now is 0.331 with a sample value [272,72], i.e., 272 cases of Class 0 from the root node of 649 and 72 cases of class 1 from the root node of 872. The Gini impurity calculated based on the right branch now is 0.435 with a sample value [377,800], i.e., 377 cases of Class 0 from the root node of 649 and 800 cases of class 1 from the root node of 872. The mean (weighted average) Gini impurity in the first depth level after the root node is $(344/1521) \times 0.331 + (1177/1521) \times 0.435 = 0.4115$, representing a Gini impurity decrease of $0.489 – 0.4115 = 0.0775$. This feature X[8] at the split of 0.032, amongst all features and their split, has the largest decrease in Gini impurity. This is thus selected as the first node (root node) decision rule.

It is seen that the splitting seems to push more class 1 to the right sub-trees. Intuitively lower operating profit margin indicates a weaker firm (class 0) and hence to the left there are now more firms in class 0 than in class 1.

On the left branch, the next decision node is based on feature 22 "Enterprise Value Multiple" – which is EV/EBITA, the total value of a company (EV) (market cap plus debts and cash) relative to its earnings before interest, taxes, depreciation, and amortization (EBITDA). Lower multiple may indicate a weaker firm (this may conversely indicate a better value for acquisition). If this multiple is <= -0.037 (recall this is standard-scaled), then the branch goes to the next level on the left with sample value [188,17] – a high proportion of class 0 versus class 1. There is a further Gini impurity reduction of 0.331 – 0.152 = 0.179. The average/mean Gini impurity decrease, or Gini importance is the (relative sample size) weighted average of the Gini impurity decreases (in absolute values) for both sides of the branching. Basically, as discussed, the decision node decides on the feature that yields the largest weighted average Gini impurity decreases.

If we use only this low depth of 3, then the testing results may not be accurate. For example, our decision rules would be (see the left-most leaf on the third level) predicting "0" (since 180 versus 12 is obviously a higher odds for "0") if X[9] <=0.031, X[22]<=-0.037, and X[8]<=0.032, and so on for the other rules. This predicted probability of class 0 is 180/192 = 93.75%. But it could be higher if more depth is used.

In code line [22], we develop the full DT without constraining the depth – letting the tree nodes expand till all leaves are pure. All features are also considered in each level of splitting with the greedy algorithm seeking the one with largest weighted average Gini impurity decrease. The DT text codes are shown in [22] and the structure of the DT is shown graphically in [25] with 22 levels of depth.

```
[22]: from sklearn.tree import DecisionTreeClassifier
      DT_model= DecisionTreeClassifier(criterion='gini')
      ### note list of default parameters used in above:
      ### max_depth = none, nodes expanded till all leaves are pure or
      ###  contain < min_samples_split_samples = 2 by default
      ### max_features = none (default) means max_features = n_features (all).
      ###  When max_features < n_features, the algorithm will select max_features
      ###  at random at each split before finding the best split among them
      ### min_impurity_decrease = 0 (default case) -- means a node will be split
      ###  if it induces impurity decrease >= this value
```

```
DT_model.fit(X_train, y_train)
y_pred_DT = DT_model.predict(X_test)
Accuracy_DT = metrics.accuracy_score(y_test, y_pred_DT)
print("DT Accuracy:",Accuracy_DT)
```

```
DT Accuracy: 0.7066929133858267
```

[26]:
```
from sklearn.metrics import confusion_matrix
confusion_matrix = confusion_matrix(y_test, y_pred_DT)
print(confusion_matrix)
```

```
[[143  72]
 [ 77 216]]
```

[27]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test, y_pred_DT))
```

```
              precision    recall  f1-score   support

           0       0.65      0.67      0.66       215
           1       0.75      0.74      0.74       293

    accuracy                           0.71       508
   macro avg       0.70      0.70      0.70       508
weighted avg       0.71      0.71      0.71       508
```

Code lines [22] to [29] show an accuracy of 70.669%, precision of 75% and recall of 74% for the case of class 1. The precision and recall for class 0 are 65% and 67% respectively. High recall in case 0 is important for this situation of being able to predict a low rating firm correctly. Predicting low rating firm to be from a high rating means a bank may lend and may suffer losses. The AUC based on class 0 is 70.12%.

5.3 Random Forest

Random Forest (RF) or Random decision trees is an ensemble (collection or group) method for classification as well as regression prediction. In a RF, many, e.g., 500, DTs are run independently (or in parallel). Each of the 500 DTs is done on a sample bootstrap aggregation (“bagging”), i.e., if the training sample is 1521, after one DT is done as seen earlier, the next DT is constructed by resampling on the 1521 sample with replacement, i.e., some cases/firms may appear more than once in a resampled sample of 1521.

Each of the trained 500 DTs gives a possibly different set of decision rules due to the resampling within the training data set and random selection of a smaller set of features. The X_train, and correspondingly X_test, i.e., the training data set, change for each re-sampling. Hence there are 500 different DTs and thus 500 different sets of decision rules. These are applied to the fixed 508 firms in y_train and y_test, i.e., the test data set.

In the testing, each of the 500 trained DTs will, using its own trained decision rules, yield its prediction vector on the test sample of the same 508 firms. Each of these prediction vector from each of the 500 DTs are possibly different.

Suppose the trained 500 DTs produce 500 vectors of the 508×1 predictions on the 508 test set firms' target labels of "0"s or "1"s. Suppose sum of predicted test cases is shown as follows.

$$\begin{bmatrix}1\\0\\0\\1\\0\\1\\0\\\vdots\\1\end{bmatrix}+\begin{bmatrix}1\\1\\0\\1\\0\\0\\0\\\vdots\\1\end{bmatrix}+\begin{bmatrix}0\\0\\0\\1\\0\\1\\1\\\vdots\\0\end{bmatrix}+\cdots\cdots+\begin{bmatrix}1\\0\\1\\0\\0\\1\\0\\\vdots\\1\end{bmatrix}=\begin{bmatrix}312\\145\\98\\436\\24\\210\\56\\\vdots\\396\end{bmatrix}_{508\times1}$$

Dividing the 508×1 vector by 500 to show the averaged "vote" or probability of being "1"s – we obtain:

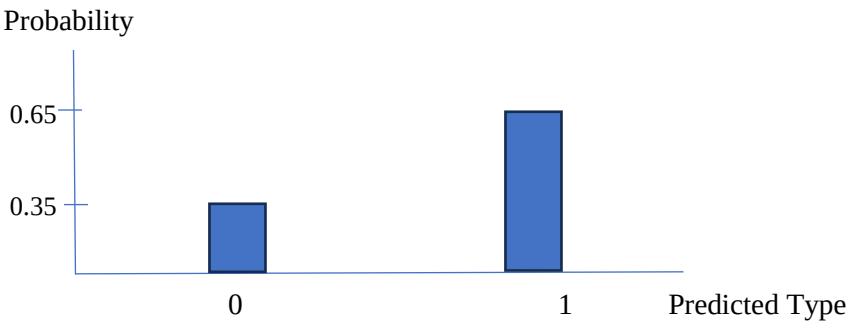
$$P=\begin{bmatrix}0.624\\0.29\\0.196\\0.872\\0.048\\0.42\\0.112\\\vdots\\0.792\end{bmatrix}$$

This means that the trained 500 DTs predicted, on averaging, based on the test set, that the first firm in the test set has predicted probability of 0.624. The second firm in the test set has predicted probability of 0.29, and so on. Using

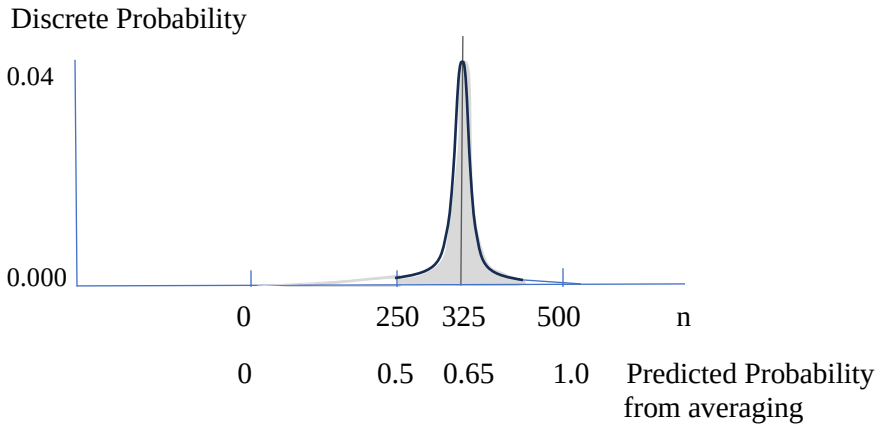
probability > 0.5 as threshold, for class "1", first firm is predicted as class "1" and second firm is predicted as class "0". Using the actual test y-value or actual firm's "1", "0" status, the confusion matrix of the RF can thus be produced, and the various prediction performance metrics can be computed. With a different threshold, e.g., 0.65, using probability > 0.65 as prediction of "1", the prediction of the first firm is "0" and that of the second firm is "0". With a threshold, e.g., 0.25, using probability > 0.25 as prediction of "1", the prediction of the first firm is "1" and that of the second firm is "1". Hence with different thresholds, and thus different corresponding confusion matrix, the ROC and its AUC can be computed.

In a RF Regression context, each predicted vector (508 × 1 vector of values) of the test set sample, based on each of the 500 trained DTs, is noted and together they are averaged to find the RF predicted probabilities (508 × 1 vector). Just like an orchestra, an ensemble produces good accuracy even if individual DTs may falter or are weak performers. The averaging can produce better results as it reduces individual DT variance in the prediction outcomes. The intuition for this is explained as follows.

Consider the single DT. After training, the probability of correctly predicting "1" from a sample point drawn from the test data set could be 0.65 (accuracy of 65%). Suppose we build a second tree, a third tree, and so on (using bagging with random resampling and random selection of subsets of features for training different trees). The accuracies of the other trees could be slightly different such as 68%, 72%, 65%, 71%, etc. We assume for illustration that all the DTs have accuracies of 65%. For the situation of a single DT for prediction, given the test sample point is type "1", the probability distribution of the likely prediction is Bernoulli, seen as follows.



When 500 DTs in a RF are considered, and assuming they are independent trees, then the number of "1"s predicted follow a binomial distribution. Let n be number of "1"s predicted and (500 – n) be the number of "0"s predicted for the same test sample point or instance of type "1". The probability of n occurring is $P(n) = {}^{500}C_n(0.65)^n(0.35)^{500-n}$ where ${}^{500}C_n = 500!/(n!\ (500-n)!)$. This distribution can be shown as follows using a normal distribution as approximation with mean of the distribution of n as 500(0.65) = 325, and variance as 500(0.65)(0.35) = 113.75 or standard deviation of 10.67.



The probability of getting an average number of votes of "1" out of 500 to be more than 50% (shown in the second X-axis) in this test sample point is Prob(n > 250) which is close to 1 (indicated by the shaded area).

Hence it is seen that whereas using a single DT can end up with a high chance of wrong prediction of 30%, the ensemble method would have a close to zero chance of wrong prediction in this case.

We continue with the credit rating accounting data set used earlier in Section 5.2. First, we apply the Random Forest method. The sample is similarly split into 25% test size and 75% training size. The following shows code lines from Chapter5-2.ipynb file.

Split the dataset: Creating a Training Set and a Test Set

- 25% of the data will be used as the test set and 75% of the data would be used to train

```
[14]: Train, Test = train_test_split(TT,
                                     test_size=0.25,
                                     random_state=0)

      X_train, y_train = Train.iloc[:,1:26], Train.iloc[:,0] ### choose the 25 features
      X_test, y_test = Test.iloc[:,1:26], Test.iloc[:,0]
```

```
[15]: X_train.shape, y_train.shape, X_test.shape, y_test.shape
```

```
[15]: ((1521, 25), (1521,), (508, 25), (508,))
```

Sklearn RandomForestClassifier is used with specification of 500 DTs (n_estimators = 500). RF method, besides bagging, also possibly randomizes by choosing a subset of the total set of 25 features for each DT. We set max_features = 'sqrt' which means that each DT randomly uses $\sqrt{25} = 5$ features of the 25 total in constructing the branching in each of the 500 DTs. Therefore, it is important to have many DTs or n_estimators. See code line[17]. The number of trees (n_estimators) and the number of features to use for each DT are two key hyperparameters in RF.

Random Forest

```
[17]: # Random Forest
      from sklearn.ensemble import RandomForestClassifier

      RF_model = RandomForestClassifier(n_estimators=500, \
                 random_state=1, max_features="sqrt", max_depth=34)
      ### default n_estimators/trees = 100; default criterion = 'gini'
      ### max_features -- The number of features to consider when
      ###  looking for the best split:
      ### default case is "sqrt"; or "log2", or None (all features),
      ###  unlike DecisionTree Classifier where default case = "none"
      ### If max_depth=None, then nodes are expanded until all leaves
      ###  are pure or until all leaves contain less than
      ### min_samples_split samples -- usu 2
      ### random_state -- resamples each of the n_estimators number of DTs
      ### bootsrapping - bagging, default=true
      ### oob_scorebool, default=False -- Whether to use out-of-bag samples
      ###  to estimate the generalization score. Only available if
      ###  bootstrap=True, i.e., if there is resampling using bootstrap.

      RF_model.fit(X_train,y_train)
      y_pred_RF = RF_model.predict(X_test)
      Accuracy_RF = metrics.accuracy_score(y_test, y_pred_RF)
      print("RF Accuracy:",Accuracy_RF)
```

```
RF Accuracy: 0.7952755905511811
```

In this investment grade/speculative grade prediction problem, the accuracy of the RF here is 79.53%. The Confusion matrix and the prediction performance classification report are shown as follows. In sklearn, the confusion matrix is usually shown with first line as positives or taking the lower y-value, y = "0" in this case.

```
[18]: from sklearn.metrics import confusion_matrix
      confusion_matrix = confusion_matrix(y_test, y_pred_RF)
      print(confusion_matrix)
```

```
[[153  62]
 [ 42 251]]
```

```
[19]: from sklearn.metrics import classification_report
      print(classification_report(y_test, y_pred_RF))
```

	precision	recall	f1-score	support
0	0.78	0.71	0.75	215
1	0.80	0.86	0.83	293
accuracy			0.80	508
macro avg	0.79	0.78	0.79	508
weighted avg	0.79	0.80	0.79	508

Compare for class Y=1, precision of 80% and recall of 86% in Random Forest versus 75% and 74% for a single DT. For class Y = 0, RF also has higher precision of 78% and higher recall of 71% compared to those of a single DT of 65% and 67% respectively.

The predicted probability of class 1 for each of the 508 test cases is given by code lines [20], [21].

```
[20]: ### Assuming your target is (0,1), then the classifier would output
      ###  a probability matrix of dimension (N,2).
      ### The first index refers to the probability that the data belong to class 0,
      ###  nd the second refers to the probability hat the data belong to class 1.
      ###  Each row sums to 1.
      ### "RF_model.predict_proba(X_test)[:,1]" will print a single array of 508
      ###  numbers - the second column or the predicted prob of test cases being 1.
      ### "RF_model.predict_proba(X_test)" prints array([[0.242, 0.758], ... for 508 pairs
```

[21]:
```
import sklearn.metrics as metrics
# calculate the fpr and tpr for all thresholds of the classification
preds_RF = RF_model.predict_proba(X_test)[:,1]
print(preds_RF)
```

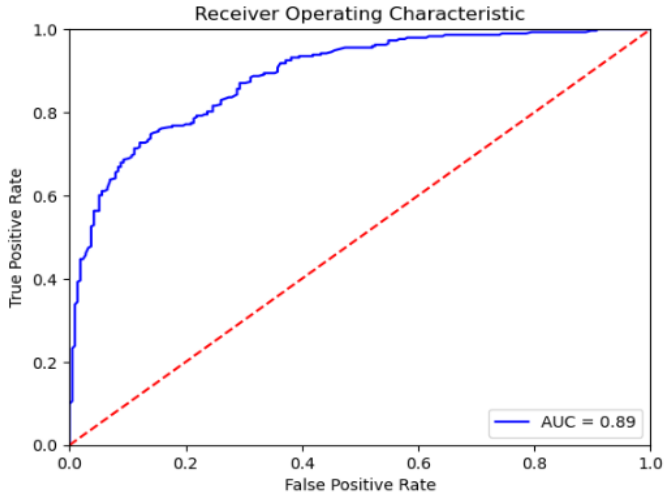
```
[0.758 0.866 0.49  0.508 0.98  0.888 0.836 0.722 0.628 0.82  0.6   0.816
 0.712 0.798 0.78  0.508 0.25  0.878 0.348 0.138 0.63  0.954 0.368 0.96
 0.522 0.634 0.752 0.76  0.204 0.23  0.794 0.36  0.354 0.734 0.636 0.156
 0.862 0.14  0.652 0.872 0.892 0.868 0.286 0.21  0.894 0.162 0.944 0.736
 0.674 0.58  0.836 0.938 0.794 0.636 0.226 0.68  0.534 0.718 0.882 0.552
 0.866 0.088 0.73  0.432 0.156 0.598 0.398 0.268 0.936 0.922 0.464 0.33
 0.834 0.968 0.096 0.386 0.796 0.452 0.846 0.872 0.966 0.046 0.94  0.774
 0.664 0.78  0.948 0.688 0.32  0.59  0.772 0.148 0.594 0.05  0.502 0.09
 ..........................................................................
```

The probabilities allow computation of the Receiver Operating Characteristic curve when the threshold for predicting the class is varied from 0 to 1. Code lines [22] to [23] show computations of the AUC in ROC.

[22]:
```
fpr, tpr, thresholds = metrics.roc_curve(y_test, preds_RF)
### matches y_test of 1's and 0's versus pred prob of 1's for each of
###  the 508 test cases
### sklearn.metrics.roc_curve(y_true, y_score,...) requires y_true as
###  0,1 input and y_score as prob inputs
### metrics.roc_curve returns fpr, tpr, thresholds (Decreasing thresholds
###  used to compute fpr and tpr)
roc_auc_RF = metrics.auc(fpr, tpr)
### sklearn.metrics.auc(fpr,tpr) returns AUC using trapezoidal rule
roc_auc_RF
```

[22]:
```
0.8868640368283197
```

[23]:
```
import matplotlib.pyplot as plt
plt.title('Receiver Operating Characteristic')
plt.plot(fpr, tpr, 'b', label = 'AUC = %0.2f' % roc_auc_RF)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```



With RF, the AUC is 88.69%. This compares well with 70.12% AUC for a single DT.

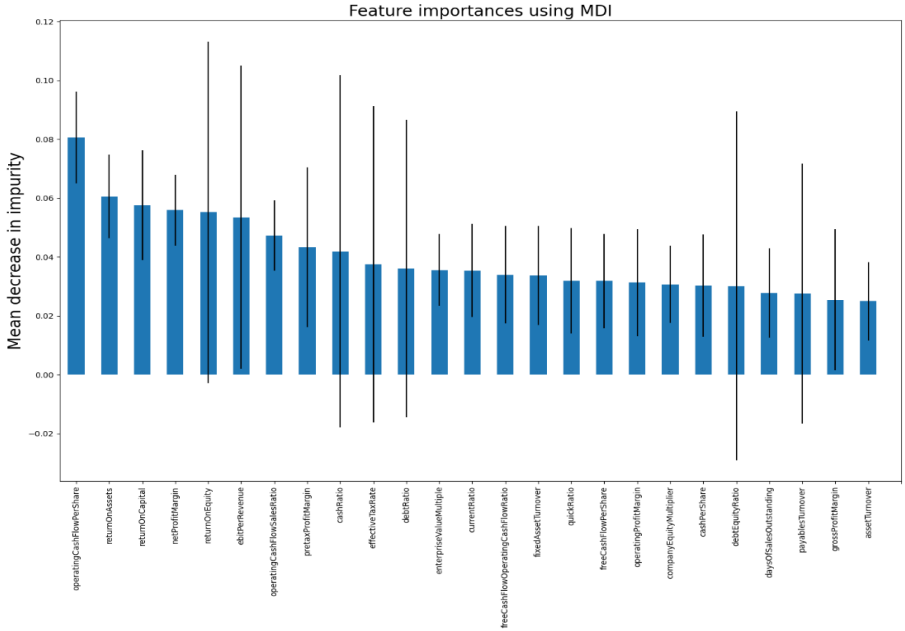
With RF, we can also analyse the "Feature Importance". For each of the 500 DTs, the features that are involved in the splitting and hence Gini importance (mean decrease in Gini impurity) from the decision node down to the next two branched nodes are noted.

Over all the 500 DTs, the total Gini importance contributed by each of the 25 features are tabulated, including contributions by the same feature more than once in a DT. This feature total is then divided by the total across the totals of the 25 features. This ratio is called the Feature Importance of the particular feature. The ratios of all features sum to 100%. The higher the % of each feature in this contribution to overall Gini importance or overall Gini weighted impurity decrease, the more important is this feature in the RF algorithm. The examining of feature importance is shown as follows in code line [25].

```
[25]: RF_model.feature_importances_  ### they sum to one

[25]: array([0.03539456, 0.03190046, 0.04191705, 0.02772411, 0.05589552,
             0.04333299, 0.02547268, 0.031284  , 0.06059753, 0.05753511,
             0.05522376, 0.02494506, 0.03374492, 0.03015326, 0.03604814,
             0.0375338 , 0.03397923, 0.03182288, 0.03024578, 0.03071357,
             0.05349687, 0.03554667, 0.0806157 , 0.0473035 , 0.02757288])
```

The feature importance output is more conveniently visualized. See code line [26].

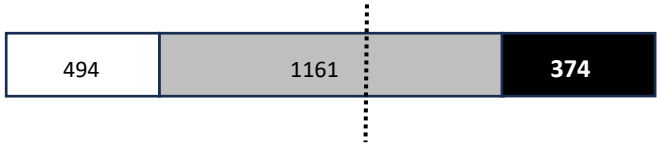
[26]:
```
import pandas as pd

forest_importances = pd.Series(importances, index=X_train.columns)
forest_importances.sort_values(ascending=False,inplace=True)

fig, ax = plt.subplots(figsize=(15,12))
forest_importances.plot.bar(yerr=std, ax=ax)
### thin line indicates 1 std err from the mean either way
###  -- doesn't mean mean decrease is neg
ax.set_title("Feature importances using MDI",fontsize=20)
### MDI is mean decrease in impurity
ax.set_ylabel("Mean decrease in impurity",fontsize=20)
ax.set(xticks=([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,
 24,25]))
### define own x-ticks to avoid clutter, entry must be a list
fig.tight_layout()

### Note: Negative feature importance value means that feature makes the loss go up.
### Either model is underfitting with not enough iteration and not enough splitting
###  use of feature or feature should be removed.
```



The output shows that in order of Feature importance, the features are Operating Cashflow per share, Return on Assets, Return on Capital, Net Profit Margin, Return on Equity, EBIT per Revenue, and so on. These are key accounting variables/ratios of the firm that largely determine the credit quality and hence if the firm is of investment grade or of speculative grade.

5.4 Multi-Class Classification

Sometimes binary classification may not be the best way to classify if the types have the following distributions where the numbers indicate the instances.



These classes are akin to the class 2 for white sector, class 1 for grey sector, and class 0 for black sector. Banks facing potential borrowing firms would try to accurately predict their unknown rating status, especially if they have bad ratings belonging to class 0 or the black sector. Banks would also like to predict if the potential client is a good firm of class 2 so that more favourable lending rates can be given to maintain longer client relationship. Hence it is more relevant to perform multi-class (3-class) prediction in this case.

We continue with the corporate ratings data set in Section 5.2 but now re-classify the firms into 3 classes, viz., Class 2 for AAA, AA, A ratings, class 1 for BBB, BB ratings, and class 0 for B, CCC, CC, C, and D ratings. The code file is Chapter5-2M.ipynb and the data set is corporate_rating3.csv. The 2029 cases are now divided into the 3 classes as follows, shown in code line [4].

```
[4]: Class
     1    1161
     2     494
     0     374
```

Standard scaling is done, as in the binary classification, to the 25 features in code line [7]. The data set is similarly divided into 75% training data and 25% test data (assuming validation phase has been done). There are 508 test cases and the distribution of these in classes 2,1,0 is shown as follows.

```
[16]: # Display the no. of cases in classes '0', '1', '2'
      pd.Series(y_test).value_counts()
```

```
[16]: Class
      1    284
      2    132
      0     92
```

The Random Forest method is performed on this multi-class prediction problem. Code line [18] shows that with ensemble of 1000 DTs, the accuracy is 69.29%.

Random Forest

```
[18]: # Random Forest
      from sklearn.ensemble import RandomForestClassifier

      RF_model = RandomForestClassifier(n_estimators=1000, random_state=1, \
                  max_features="sqrt", max_depth=34, criterion="entropy")

      RF_model.fit(X_train,y_train)

      y_pred_RF = RF_model.predict(X_test)
      Accuracy_RF = metrics.accuracy_score(y_test, y_pred_RF)
      print("RF Accuracy:",Accuracy_RF)
```

```
RF Accuracy: 0.6929133858267716
```

Code lines [21] and [22] show the computed confusion matrix and the classification report for this multi-class prediction.

```
[21]: from sklearn.metrics import confusion_matrix
      confusion_matrix_RF = confusion_matrix(y_test, y_pred_RF)
      print(confusion_matrix_RF)
```

```
[[ 36  54   2]
 [ 12 249  23]
 [  0  65  67]]
```

The first, second, and third row (from top to bottom) in the confusion matrix denotes actual class 0, 1, and 2 test instances. The first, second, and third columns (from left to right) denotes the predicted class 0, 1, and 2 respectively. The recall for class 0 is 36/(36+54+2) = 39.1%. The precision for class 0 is

36/(36+12+0) = 75%. The recall for class 2 is 67/(67+65+0) = 50.75%. The precision for class 2 is 67/(67+23+2) = 72.8%.

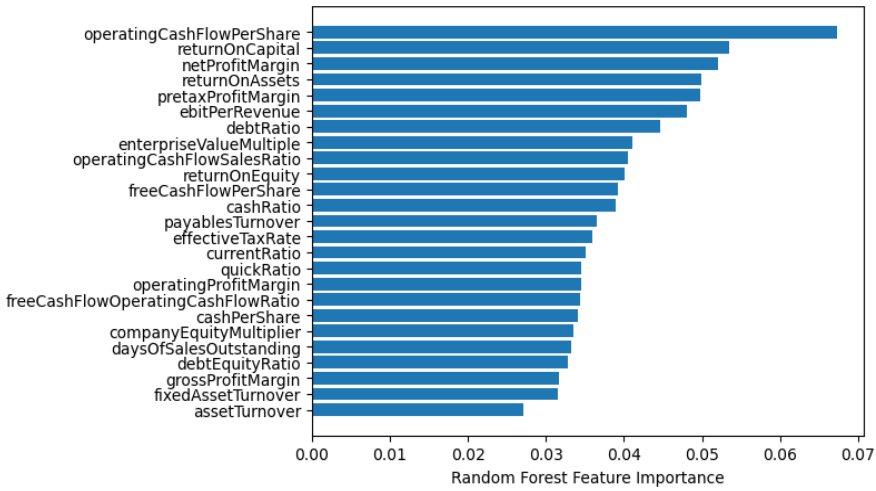
```
[22]: from sklearn.metrics import classification_report
      print(classification_report(y_test, y_pred_RF))
```

	precision	recall	f1-score	support
0	0.75	0.39	0.51	92
1	0.68	0.88	0.76	284
2	0.73	0.51	0.60	132
accuracy			0.69	508
macro avg	0.72	0.59	0.63	508
weighted avg	0.70	0.69	0.68	508

The recall for class 0 appears low and can be improved via more effective algorithms.

The feature importance is computed via code line [26] in a different presentation as follows. They are largely similar to those in the binary classification.

```
[26]: ### another representation
      sorted_idx = RF_model.feature_importances_.argsort()
      plt.barh(X_train.columns[sorted_idx],RF_model.feature_importances_[sorted_idx])
      plt.xlabel("Random Forest Feature Importance")
      plt.figure(figsize=(20, 15), dpi=80)
      fig.tight_layout()
```



For multi-class classification, it is not a natural step to construct a single ROC curve since the ROC curve involves the true positive rate plot versus the false positive rate and these come more naturally from a binary classification table. However, we can construct 3 separate ROCs, one for each class as the positive case.

Code line [27] shows how to pull out the predicted probability vectors 'preds_RF[:,0]', 'preds_RF[:,1]', 'preds_RF[:,2]' for class 0, 1, and 2 respectively. Each vector gives the probability of class in the 508 instances.

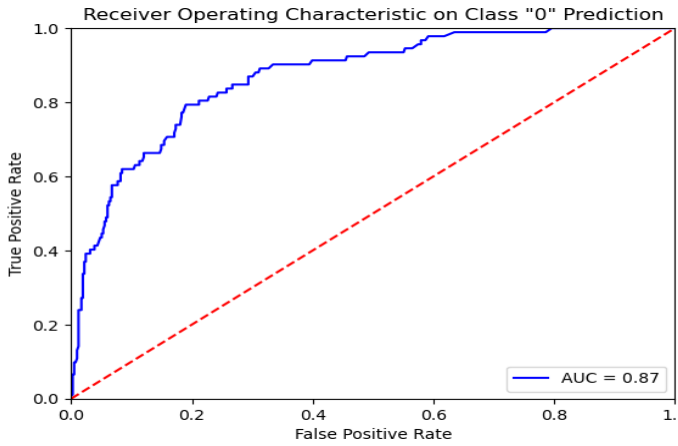
```
[27]: import sklearn.metrics as metrics
      preds_RF = RF_model.predict_proba(X_test)
      ### Target here is ('0','1','2'). RF_model.predict_proba(X_test) produces
      ###  508 x 3 matrix of 3 cols. First col preds_RF[:,0] row i gives
      ###  predicted prob that case i belongs to class '0'
      ###  Second col preds_RF[:,1] row i gives predicted prob that case i
      ###  belongs to class '1'. Third col preds_RF[:,2] row i gives predicted
      ###  prob that case i belongs to class '2'. All elements in row i,
      ###  i.e., preds_RF[i,0] + preds_RF[i,1] +preds_RF[i,2] = 1
      ### We need these probs to compare against varying thresholds to give ROC

      preds_RF_0 = preds_RF[:,0]
      preds_RF_1 = preds_RF[:,1]
      preds_RF_2 = preds_RF[:,2]
```

Code line [28] binarizes each value of either "1", "2", or "3" in y_test to 3 columns. In the first column, the value is "1" if the test instance is of class 0, otherwise it is "0". In the second column, the value is "1" if the test instance is of class 1, otherwise it is "0". In the third column, the value is "1" if the test instance is of class 2, otherwise it is "0". Thus 'yy' is a 508 × 3 matrix.

```
[28]: from sklearn.preprocessing import label_binarize
      yy = label_binarize(y_test, classes=[0,1,2])
      ### yy[:,0] is 508 x 1 where kth row = 1 if it belongs to class 0, o.w. 0
      ### yy[:,1] is 508 x 1 where kth row = 1 if it belongs to class 1, o.w. 0
      ### yy[:,2] is 508 x 1 where kth row = 1 if it belongs to class 2, o.w. 0
```

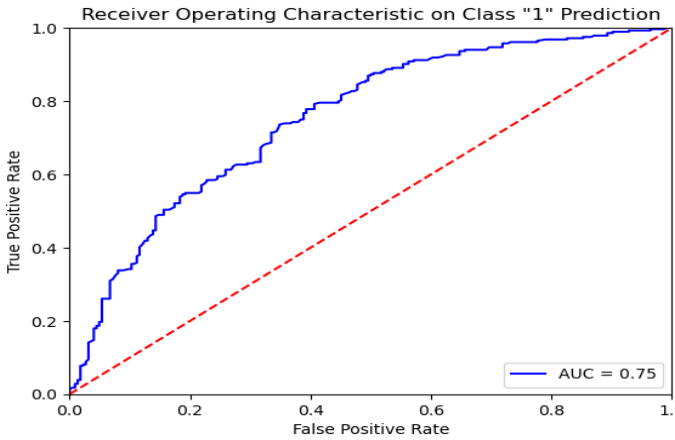
To construct the ROC for class 0, we match the binarized 'yy[:,0]' (first column) with 'preds_RF[:,0]', the predicted probability that each row instance is in class 0. This is shown in code line [29] and the ROC is plotted as follows. The area under the ROC or AUC is 87.0%.

```
[29]: import sklearn.metrics as metrics
      ### calculates the fpr and tpr for all thresholds of the classification 0
      ###  prediction on y_test
      fpr_0, tpr_0, thresholds = metrics.roc_curve(yy[:,0], preds_RF_0)
      ### matches y_test of 1's and 0's versu pred prob of 1's for each of the 508
      ###  test cases/instances for '0' class
      ### sklearn.metrics.roc_curve(y_true, y_score,...) requires y_true as 0,1 input
      ###  and y_score as prob inputs
      ### metrics.roc_curve returns fpr, tpr, thresholds (Decreasing thresholds used
      ###  to compute fpr and tpr)
      roc_auc_RF_0 = metrics.auc(fpr_0, tpr_0)
      ### sklearn.metrics.auc(fpr,tpr) returns AUC using trapezoidal rule
      roc_auc_RF_0

[29]: 0.8695521530100334
```



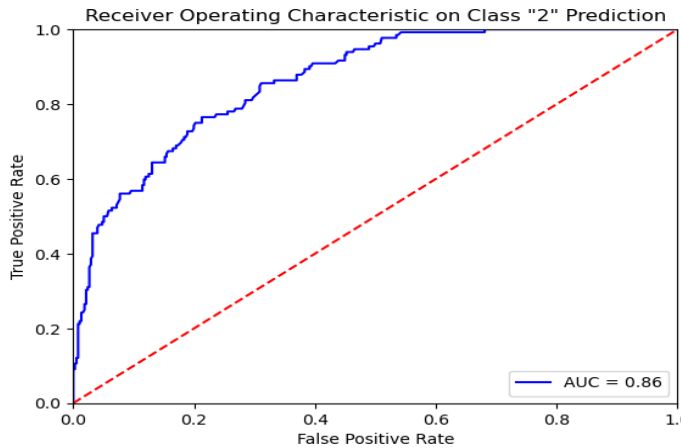
To construct the ROC for class 1, we match the binarized 'yy[:,1]' (first column) with 'preds_RF[:,1]', the predicted probability that each row instance is in class 1. This is shown in code line [31] and the ROC is plotted as follows. The area under the ROC or AUC is 75.4%.

```
[31]: import sklearn.metrics as metrics
      ### calculate the fpr and tpr for all thresholds of the classification 1
      ###  prediction on y_test
      fpr_1, tpr_1, thresholds = metrics.roc_curve(yy[:,1], preds_RF_1)
      roc_auc_RF_1 = metrics.auc(fpr_1, tpr_1)
      roc_auc_RF_1

[31]: 0.7543228118712275
```



Finally, to construct the ROC for class 2, we match the binarized 'yy[:,2]' (third column) with 'preds_RF[:,2]', the predicted probability that each row instance is in class 2. This is shown in code line [33] and the ROC is plotted as follows. The area under the ROC or AUC is 86.4%.

```
[33]: import sklearn.metrics as metrics
      ### calculate the fpr and tpr for all thresholds of the classification 2
      ###  prediction on y_test
      fpr_2, tpr_2, thresholds = metrics.roc_curve(yy[:,2], preds_RF_2)
      roc_auc_RF_2 = metrics.auc(fpr_2, tpr_2)
      roc_auc_RF_2

[33]: 0.8635658446163765
```



5.5 Concluding Thoughts

Decision Trees are a non-parametric method to predict the type or label of a case or instance based on the features that come with the instance. In this chapter, we see how the accounting features such as accounting and financial ratios of a firm can be used to predict the credit rating of the firm.

When the ensemble approach is taken as in the Random Forest algorithm, RF reduces the variance of outcomes in a single DT. RF also randomizes selection of features. RF thus handles high dimensionality well as it can reduce use of all features; it also handles missing values since any missing value amounts to having not selected that feature for that row. However, a RF prediction outcome is also harder to explain or interpret since there is no fixed set of features and weights as it is a collection of different DTs.

We also see how in the problem of a bank predicting a firm's credit rating and thus the willingness to lend, it is more relevant to perform multi-class (3-class) prediction in this case. It is more important to predict low credit-rating firms and avoid such loans to reduce potential default loss. Secondarily, it is also important to predict high credit-rating firms and encourage such loans to build up a healthy pipeline of lending profits.

There is another popular ensemble technique to create an ensemble or collection of predictors. The idea is also to construct many DTs – but the difference is that the target variable for each subsequent DT is different. The idea with each subsequent DT is to solve for the net error left from the past DTs and to build stronger and more accurate DT starting from weaker ones. The latter technique is called gradient boosting and is covered in the next chapter.

References

Aurelien Geron, (2019), "Hands-on Machine Learning with Scikit-Learn, Keras & TensorFlow", 2nd ed., O'Reilly Publisher.

Marcos Lopez De Prado (2018), "Advances in Financial Machine Learning", Wiley.

https://medium.com/the-artificial-impostor/feature-importance-measures-for-tree-models-part-i-47f187c1a2c3