# **Introduction to Decision Trees**

As the name goes, a decision tree uses a tree-like model to make predictions. It resembles an upside-down tree. It is also very similar to how you make decisions in real life: you ask a series of questions to arrive at a decision.

A decision tree splits the data into multiple sets. Then, each of these sets is further split into subsets to arrive at a decision.

|  |  |
| --- | --- |
| https://cdn.upgrad.com/UpGrad/temp/81f1f135-b98b-4144-802e-6a81c4bf9af4/1.png | On each node, you ask a question to further split the data held by the node. If the test passes, you go left; otherwise, you go right.  In the heart disease example shown above, the leaf nodes (bottom) are labelled **yes** (the person has heart disease) or **no**(the person doesn’t have heart disease).  The decision tree predicts that if, ‘Thal’ is not equal to 3, ‘coloured fluoroscopy’ is less than 0.5, ‘exercise.angina’ is equal to 0, and ‘age’ is less than 51, then the person will have heart disease (the path indicated by the red line in the figure). Similarly, there are three other paths which lead to a yes labelled leaf. |

**Interpreting a Decision Tree**

Now if a model predicts that a data point belongs to class A, how do you figure out which attributes were the most important predictors? Decision trees make it very easy to determine the important attributes.  If a test splits the data into more than two partitions, this is called a **multiway decision tree.**

The decision trees are easy to interpret. Almost always, you can identify the various factors that lead to the decision. In fact, trees are often underestimated for their ability to relate the predictor variables to the predictions. As a rule of thumb, if interpretability by laymen is what you're looking for in a model, decision trees should be at the top of your list.

|  |  |
| --- | --- |
|  |  |

So the decision trees can go back and tell you the factors leading to a given decision. In SVMs, if a person is diagnosed with heart disease, you cannot figure out the reason behind the prediction. However, a decision tree gives you the exact reason, i.e. either 'Thal is 3, the pain type is neither 1, nor 2, nor 3, and the coloured fluoroscopy is greater than or equal to 0.5', or 'Thal is not equal to 3, and either of the three tests, shown in the right half of the tree, failed'.

Consider the heart disease decision tree again. Given that a patient is diagnosed with heart disease, you can easily trace your way back to the multiple tests that would have led to this diagnosis. One such case could be where the patient doesn’t have thal = 3, and coloured fluoroscopy is greater than or equal to 0.5.

In other words, each decision is reached via a path that can be expressed as a series of ‘if’ conditions satisfied together, i.e., **if** ‘thal’ is not equal to 3, and **if** coloured fluoroscopy is greater than or equal to 0.5, then the patient has heart disease. Final decisions in the form of class labels are stored in leaves.

|  |  |
| --- | --- |
|  | In regression problems, a decision tree splits the data into multiple subsets. The difference between decision tree classification and decision tree regression is that in regression,**each leaf represents a linear regression model**, as opposed to a class label.    In this module, you won't study decision tree regression in detail, but only decision tree classification because that is what you’ll most commonly work on. However, remember that if you get a data set where you want to perform linear regression on multiple subsets, decision tree regression is a good idea.  **Additional Readings:**   * [Decision Trees scikit-learn documentation](http://ogrisel.github.io/scikit-learn.org/sklearn-tutorial/modules/tree.html) |

**Regression**

Select all that is correct about decision tree classification and regression models.

Top of Form



Leaves in classification contain labels.

**Feedback :**In classification, the target variable is discrete. Hence, each data point in a leaf has an associated class label.

**Correct**



Leaves in regression contain labels.



Leaves in classification contain models.



Leaves in regression contain models.

**Feedback :**True. Each leaf in regression contains a model that is used to make predictions.

Bottom of Form

**Decision trees**

Say you have a data set with lots of categorical variables and some numerical variables. The target variable is continuous, so it’s a regression problem. After some exploratory data analysis, you figure out that it will be best to perform decision tree regression instead of linear regression. Which of the following statements will be correct? More than one option may be correct.

Top of Form



It is hard to represent all the data via a single model; so you don’t want to use the linear regression model.

**Feedback :**Decision tree regression is performed because the entire data set cannot be represented by a single linear regression model.

**Correct**You missed this!



Decision trees will split the data set into multiple sets and will apply linear regression to each set separately.

**Feedback :**Leaves obtained after splitting contain linear regression models to make predictions.

**Correct**



**Algorithms for Decision Tree Construction**Bottom of Form

# **Concept of Homogeneity**

Given 10 attributes, how do you decide which attribute to split on first? Should you randomly select an attribute and split the data set on it? Or should there be a selection criterion to choose a particular attribute ahead of the other nine attributes?

|  |  |
| --- | --- |
|  | If a partition contains data points with identical labels (say, label 1), then you can classify the entire partition as label 1. But this is a rather oversimplified example. In real-world data sets, you will almost never get completely homogenous data sets (or even nodes after splitting). So try to do the best you can, i.e. try to split the nodes such that the resulting nodes are **as homogenous as possible.**  If the node is blood pressure, try and split it with a rule such that all the data points that pass the rule have one label (i.e. is as homogenous with as possible), and those that don't, have another. Always try to generate partitions that result in homogeneous data points. For classification tasks, a data set is **completely** **homogeneous**if it **contains only a single class label**. |

The aim of decision tree splitting is to Increase Homogeneity. More homogeneity will mean that most of the data points in the set belong to the same class label. Hence, classifying all the data points of that set, to get them to belong to that class, will result in lesser errors.

|  |  |
| --- | --- |
|  | So you go step-by-step, picking an attribute and splitting the data such that the homogeneity increases after every split. You stop splitting when the resulting leaves are sufficiently homogenous. What is sufficiently homogenous? Well, you define the amount of homogeneity which, when achieved, the tree should stop splitting further.  To reach maximum homogeneity the data set needs to be split recursively. In each layer we check can we perform some test against some homogeneity threshold.  A split that gives you a homogenous subset is much more desirable than the one that results in a 50-50 distribution (in the case of two labels). All the data points belong to one label in a completely homogeneous set. |

Out of so many attributes, a decision tree calculates the improvement in homogeneity associated with each attribute and picks the one that results in the maximum increase in homogeneity.

# **Gini Index**

There are various ways to quantify homogeneity, such as the **Gini index, information gain, entropy**, etc.

Consider the dataset of female and male employees of the company shown in the lecture. There are 1000 employees in total. Let's say, you split the data on gender. On splitting, you get two nodes with 500 observations each, call them male-node and female-node. In the male-node, 300/500 play football. In the female node, only 10/500 play football. Gini index of male-node = P(play football)^2 + P(doesn't play football)^2 = (0.6)^2 + (0.4)^2 = 0.52. What is the Gini index of female-node?

(Note: The Gini index = ∑ki=1p2i​, where pi is the probability of finding a point with the label *i*, and *k* is the number of different labels.)

(0.02)^2 + (0.98)^2

**Feedback :***The probability that a female plays football = 0.02 and the probability that a female doesn't play football = 0.98.*

**Split on gender**

What is the Gini index of the partitions if you split on ‘age’?

|  |  |
| --- | --- |
| https://images.upgrad.com/82eeffaa-5748-48f9-8927-c94ccc77a888-DT16.PNG | 0.7∗[(260/700)2+(440/700)2]+0.3∗[(50/300)2+(250/300)2]  **Feedback :**  *The Gini index of the partition with age < 50 = (260/700)^2 + (440/700)^2 . The Gini index of the partition with age > 50 = (50/300)^2 + (250/300)^2. The Gini index of all partitions = 0.7\*(Gini index of partition with age < 50) + 0.3\*(Gini index of partition with age > 50)* |

Assume that you have a data set with 2 class labels. If the data set is completely homogeneous (all the data points belong to label 1), then the probability of finding a data point corresponding to label 2 will be 0 and that of label 1 will be 1. So p1= 1, and p2= 0. The Gini index, equal to 1, will be the highest in such a case. The higher the homogeneity, the higher the Gini index.

|  |  |
| --- | --- |
|  | ‘Gender’ is a better attribute to split on as it yields a higher value of Gini index as compared to ‘age’. This means that gender gives a better split that helps distinguish between football players and non-football players. This is intuitive as well: splitting on gender is expected to be more informative than age; this is because football is usually more popular among males.    You had to make a choice between two splits: age and gender. The Gini index of splitting on gender is higher than that of splitting on age; so you go ahead and split on gender. |

What is the Gini index if all the data points in a data set have the same label?

The probability of exactly one class will be 1, and the probability of all the other classes will be 0. So, the Gini index, which is given by ​*∑*ki=1p2i​, will be 1.

Given a data set, calculate the Gini index if 50% of the data points belong to label 1, and the other 50% belong to label 2.

Top of Form



The Gini index = (12)2+(12)2.

**Feedback :**p1*= 0.5, and*p2*= 0.5. So the Gini index, which is given by ∑*ki=1p2i*, will be*(12)2+(12)2*.*

When is the Gini index maximum ?

**Feedback :**When all the points in the data set belong to one class label leading to the maximum homogeneity, the Gini index will be maximum.

# **Entropy and Information Gain**

We will now look at one more homogeneity measure: **information gain**. The idea is to use the notion of entropy. Entropy quantifies the degree of disorder in the data, and like the Gini index, its value also varies from 0 to 1. If a data set is **completely homogenous**, then the **entropy of such a data set is 0**, i.e. there’s no disorder. The entropy is given by

**ε[D]=−∑ki=1pilog2pi,**

where pi  is the probability of finding a point with the label i, k is the number of different labels, and ε[D] is the entropy of the data set D.

* Gini index is a measure for homogeneity
* Entropy is a measure of chaos, disorderness
* Entropy is maximum when Gini index is minimum
* Entropy is zero when any Pi = 1 and everything else is zero
  + If one of the Pi = 1, then that means all the labels are identical hence there is no chaos so entropy is zero

|  |  |
| --- | --- |
|  | Where ε[D] is the entropy of the parent set D (data before splitting), ε[DA] is the entropy of the partitions obtained after splitting on attribute A, ε[DA=i] is the entropy of the partition where the value of attribute A for the data points is i, DA=i is the number of data points for which the value of attribute A is i, and k is the number of different labels.  This gives us the gain if we split the original data set D, on attribute A. Now to compute the best possible split all we need to do is try doing this for every attribute A and for each attribute we compute the gain (D,A) and pick the attribute which gives us maximum information gain. |

**how we can decide the threshold in Decision trees?**

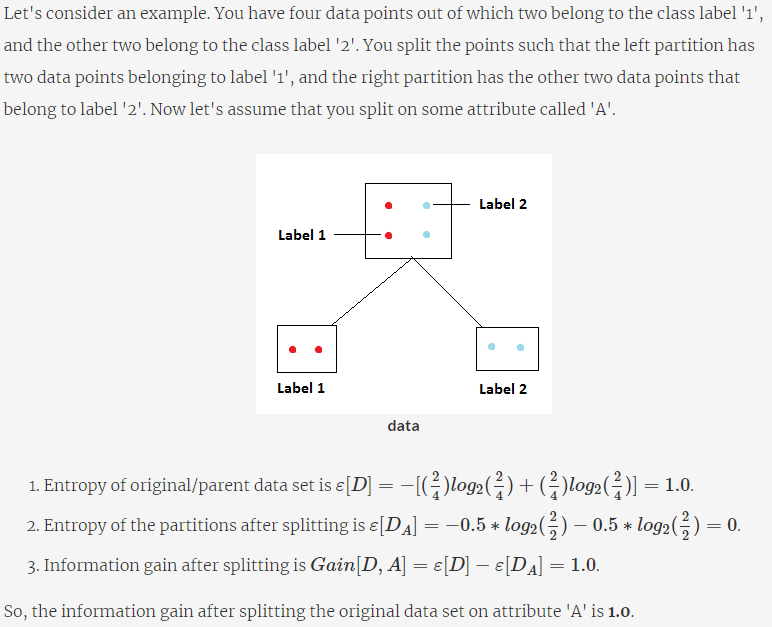
The algorithm for example python sklearn DecisionTreeClassifier() library will try out each value in turn for the particular attribute as the threshold. Then the one value which maximizes information gain (or other measure) which was in the dataset is chosen. This value is then compared with the other attributes' information gain value and then the highest value is chosen.

Decision tree uses entropy and information gain to select feature which gives the best split. Intuitively, we can say thet best split is when we can separate the classes accurately based on that feature. Therefore, the attribute with the maximum Information Gain is chosen to be the best split.

**Choosing Hyperparameter using Grid Search**

In grid search you pass a possible list of values for the different hyperparameters that you are trying to tune. For example- n\_estimators = [1, 2, 3, 4, 5] and min\_split = [10, 20, 30].

Now, you train your model and say you get n\_estimators = 5 and min\_split = 20. Now, these are optimal values among the possible set of values that you passed initially, but they might not be global optimal values. There is a possibility that maybe n\_estimators = 10 and min\_split = 15 are the best possible values but this is something which you learn by some experiment or you just expand the list of possible values and keep on trying.



When is the information gain maximum? (Select the most appropriate option.)

Top of Form



When the decrease in entropy, from the parent set to the partitions obtained after splitting, is maximum.

**Feedback :***The information gain is equal to the entropy change from the parent set to the partitions. So it is maximum when the entropy of the parent set minus the entropy of the partitions is maximum.*

Bottom of Form

|  |  |
| --- | --- |
| https://images.upgrad.com/82eeffaa-5748-48f9-8927-c94ccc77a888-DT16.PNG | Recall that the **entropy** is given by ε[D]=−∑ki=1pilog2pi, where pi  is the probability of finding a point with the label i, k is the number of different labels, and ε[D] is the entropy of the data set D.    **The information gain** is defined as Gain(D,A)=ε[D]−∑ki=1(DA=i/D)∗ε[DA=i], where ε[D] is the entropy of the parent set D (data before splitting), ε[DA=i] is the entropy of the partition where the value of attribute A for the data points is *i*, DA=i is the number of data points for which the value of attribute A is *i*, and *k* is the number of different labels. |

Details of the data set: There are 1000 employees in an organisation, of which 500 are females and 500 are males. The number of employees below 50 years of age is 700 and above 50 years is 300. Given the current data set of these 1000 employees, you want to predict whether in the future, a given employee will play football or not. Here,

**P** implies ‘plays football’ - class A = **label 1.**

**N** implies ‘does not play football’ - class B = **label 2**.

A total of 10 females and 300 males play football in the organisation. 260 people who are less than 50 and 50 people above the age of 50 play football, as shown in the figure above.

Considering the above data set, which is not homogeneous, you want to split the data such that the resulting partitions are as homogenous as possible. This is a classification problem, and there are two output classes or labels. Here, you’re using information gain as the homogeneity measure. While making your first split, you want to choose an attribute such that the information gained is maximum.

Calculate the homogeneity of the given data set using entropy.

Top of Form

−(310/1000)∗log2(310/1000)−(690/1000)∗log2(690/1000)

**Feedback :***Entropy =*−(p1)∗log2(p1)−(p2)∗log2(p2)*. Here,*p1=310/1000*, and*p2=690/1000*.*

Bottom of Form

What is the entropy of the partitions if you split on ‘gender’?

0.5∗[−(10/500)∗log2(10/500)−(490/500)∗log2(490/500)]+0.5∗[−(300/500)∗log2(300/500)−(200/500)∗log2(200/500)]

**Feedback :***Entropy of partitions = (fraction of females)\*(entropy of partition with all females) + (fraction of males)\*(entropy of partition with all males).*

*Entropy of females =*−(10/500)∗log2(10/500)−(490/500)∗log2(490/500)*; and entropy of males =*−(300/500)∗log2(300/500)−(200/500)∗log2(200/500)*.*

*The fraction of females = 0.5; and the fraction of males = 0.5.*

What is the information gain if you split the original data set on ‘gender’?

Entropy of original data set - entropy of partitions obtained after splitting on ‘gender’ = 0.89317 - 0.5562.

**Feedback :***Calculate the entropy of the original data set, and subtract the entropy of the partitions obtained after splitting, to get the information gain.*

The information gain, if you split on ‘age’, is 0.031929, and it is 0.33697 if you split on ‘gender’. What attribute should you split the original data on?

Top of Form

Gender

**Feedback :***You split on the attribute that maximises the information gain. The information gain on gender is greater than the information gain on age.*Bottom of Form

# **Splitting by R-squared**

In decision tree linear regression we can calculate R2 for every split and se the goodness of fit is there or not. If the R2 value is high enough then we stop there and that split becomes a leaf. If not, then we choose any other attribute and try to split to check the homogeneity. The *R2* is used to measure the homogeneity in regression, where the target variable is continuous. So split the data such that the R2 of the partitions obtained after splitting is greater than that of the original or parent data set. In other words, the fit of the model should be as ‘good’ as possible after splitting. In case of regression model

* Each leaf in regression contains a model that is used for prediction.
* R2 gives a sense of how good or bad the linear regression fit is.
* If the R2 is high enough, then the data is not split further.
* Sometimes a single linear regression model is not good enough to perform the regression task, so you split the data into smaller chunks and assign one linear regression model to each chunk. As you saw in the lecture, people in different age brackets should be represented using different models.
* Decision tree regression and classification are similar in the sense that both try to pick an attribute (for splitting) that maximises the homogeneity of the data set. A decision tree splits the data set on that attribute which results in the maximum increase in homogeneity.

**The steps involved in decision tree construction. Arrange the following steps in the order of their occurrence:**

1. You have a data set, D, with categorical as well as numerical attributes and continuous target variables. So it is a regression problem. Hence, you apply decision tree regression to it.
2. Select a homogeneity measure for splitting. Since it is regression, let’s choose R2.
3. After selecting the homogeneity measure, you need to decide the first attribute to split the original data set, D, on.
4. Split ‘D’ on all the attributes one by one, and select the attribute that results in the maximum increase in homogeneity after splitting.
5. Split the original data set, D, on the selected attribute.
6. Keep on splitting the subsequent data sets till you get a sufficiently high R2.
7. Now that the R2 is sufficiently high, stop splitting.
8. Each leaf will now represent a linear regression model.

Decision tree regression model can be studied from Internet available material. The decision tree classification mostly used in industry.

# **Advantages and Disadvantages**

Decision trees are very intuitive and promising algorithms for dealing with categorical attributes. Most of the other ML algorithm does not able to handle the categorical variable naturally. They require to translate the categorical variable in numbers by means dummies or some other. But for decision trees there is no such limitations. We can go ahead and build a model with any kind of data set without any problem.

To summarise the advantages,

* Predictions made by a decision tree are easily interpretable.
* A decision tree does not assume anything specific about the nature of the attributes in a data set. It can seamlessly handle all kinds of data — numeric, categorical, strings, Boolean, and so on.
* It does not require normalisation since it has to only compare the values within an attribute.
* Decision trees often give us an idea of the relative importance of the explanatory attributes that are used for prediction.

The decision tree has major disadvantages as it tends to overfit the data.

|  |  |
| --- | --- |
|  | Here, we can see the tree has mug up the entire training data. This model is not going to applicable to anything else outside the training data. Every leaf of the tree will represent one single data point in the training data.  A model which is trying to be overfit is also tending to be unstable. In Bias/Variance trade-off a model which tries to reduce the bias too much meaning it tries to bring the error term down to zero, in the process very likely to become too complex. A complex model would have many variances which means it becomes extremely sensitive to a very minor changes in the data points. |

To summarise the disadvantages,

* Decision trees tend to overfit the data. If allowed to grow with no check on its complexity, a tree will keep splitting till it has correctly classified (or rather, mugged up) all the data points in the training set.
* Decision trees tend to be very unstable, which is an implication of overfitting. A few changes in the data can change a tree considerably.

Suppose you are getting an accuracy of 40% on the test data and 98% on the training data.

Top of Form

The model has a low variance and high bias.

The model has a high variance and a low bias.

**Feedback :**The model has memorised the data, giving you a 98% training accuracy and leading to a high variance. Since it can now represent the training set very well, it has a low bias.

**Correct**

The model is overfitting.

**Feedback :**The test accuracy is very low (40%). The model is unable to work well on unseen/test data. It has memorised the training set. Hence, it is overfitting.Bottom of Form

# **Tree Truncation**

Earlier, you saw that decision trees have a strong tendency to overfit the data, which is a serious problem. So you have to keep an eye on the size of the tree. A very large tree will have a leaf just to cater to a single data point.

There are two broad strategies to control overfitting in decision trees: **truncation and pruning.** There are two ways to control overfitting in trees:

1. **Truncation** - Stop the tree while it is still growing so that it may not end up with leaves containing very few data points. This is also called **pre-pruning**.
2. **Pruning** - Let the tree grow to any complexity. Then, cut the branches of the tree in a bottom-up fashion, starting from the leaves. It is more common to use pruning strategies to avoid overfitting in practical implementations.

The truncation policy is essentially trial-n-error. The general methods of truncation are as follows:

1. Limit the minimum size of the partition after a split
2. Minimize change in the measure of homogeneity
3. Limit the depth of the tree
4. Set a minimum threshold on the number of samples that appear in a leaf
5. Set a limit on the maximum number of leaves present in tree

Though there are various ways to truncate or prune trees, the **DecisionTreeClassifier** function in **sklearn** provides the following hyperparameters which you can control:

1. **criterion (Gini/IG or entropy):** It defines the function to measure the quality of a split. Sklearn supports “gini” criteria for Gini Index & “entropy” for Information Gain. By default, it takes the value “gini”.
2. **max\_features:** It defines the no. of features to consider when looking for the best split. We can input integer, float, string & None value.
   1. If an integer is inputted then it considers that value as max features at each split.
   2. If float value is taken then it shows the percentage of features at each split.
   3. If “auto” or “sqrt” is taken then max\_features=sqrt(n\_features).
   4. If “log2” is taken then max\_features= log2(n\_features).
   5. If None, then max\_features=n\_features. By default, it takes “None” value.
3. **max\_depth:** The max\_depth parameter denotes maximum depth of the tree. It can take any integer value or None. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min\_samples\_split samples. By default, it takes “None” value.
4. **min\_samples\_split:** This tells above the minimum no. of samples required to split an internal node. If an integer value is taken then consider min\_samples\_split as the minimum no. If float, then it shows percentage. By default, it takes “2” value.
5. **min\_samples\_leaf:** The minimum number of samples required to be at a leaf node. If an integer value is taken then consider - -min\_samples\_leaf as the minimum no. If float, then it shows percentage. By default, it takes “1” value.

**Comprehension - Truncation and Pruning**

The process of splitting only when there is a sufficient number of data points in the node is called \_\_\_\_\_\_.

Top of Form

**Truncation**

**Feedback :**

*Truncation lets you split the data only when the number of data points in the node is greater than or equal to the 'minsplit'.*

Bottom of Form

Which hyperparameter controls the minimum no. of samples required to split an internal node?

Top of Form

**min\_samples\_split**

**Feedback :**

*The min\_samples\_split specifies the minimum number of data points a node should have for it to be considered for splitting.*

Bottom of Form

\_\_\_\_\_\_\_\_\_ takes care of the minimum number of samples required to be at a leaf node.

Top of Form

min\_samples\_split

**min\_samples\_leaf**

**Feedback :**

*The minimum number of samples required to be at a leaf node. If an integer value is taken then consider - -min\_samples\_leaf as the minimum no. If float, then it shows percentage. By default, it takes “1” value.*

Assume that you have set the min\_samples\_leaf as 3 and the min\_samples\_split as 6. Consider a node with 10 data points. On splitting on an attribute, one leaf gets 2 points, and the other one gets 8 data points. This split will not be executed. Why?

Top of Form

The number of data points in the node > min\_samples\_split.

**The number of data points in one of the leaves < min\_samples\_leaf.**

**Feedback :**

*The number of data points in one of the leaves is 2, which violates the condition that the number of data points in a leaf should be at least 3 (as specified by the min\_samples\_leaf).*

Bottom of Form

Bottom of Form

# **Tree Pruning**

Tree pruning is a method of chopping off parts of a tree once it is fully grown. It is a bottom-up approach used to solve the problem of overfitting. One way to do this is called **Reduced error pruning**.

|  |  |  |
| --- | --- | --- |
|  | In pruning, you chop off the tree branches; this results in a decrease in tree complexity. It also helps in reducing overfitting.  Before proceeding, please note that the data set is divided into three parts: the training set, the validation set and the test data. The validation set is used to tune hyperparameters, i.e. after deciding on a set of hyperparameters for a tree, you check the accuracy of the tree on the validation set. | |
|  | | You check the performance of a pruned tree on a validation set. If the accuracy of the pruned tree is higher than the accuracy of the original tree (on the validation set), then you keep that branch chopped. Remember that the validation set is the third part of the data set, the first and second being the training and test set. |

If the accuracy after pruning on the unseen data decreases significantly, then

Top of Form

You should prune the branch.

**Feedback :***The accuracy is decreasing after pruning, so you wouldn't prune the branch.*

**Incorrect**

You shouldn’t prune the branch.

**Feedback :***You should prune only if the accuracy after pruning does not decrease.*

Bottom of Form

# **Python Sklearn Decision Tree classifier**

﻿

## **Data Preparation**

There are a number of preprocessing steps we need to do before building the model. Firstly, note that we have both categorical and numeric features as predictors. In previous models such as linear and logistic regression, we had created **dummy variables** for categorical variables, since those models (being mathematical equations) can process only numeric variables.

All that is not required in decision trees, since they can process categorical variables easily. However, we still need to **encode the categorical variables** into a standard format so that sklearn can understand them and build the tree. The sklearn library requires to encode the **categorical variables** into **numeric labels**. We'll do that using the LabelEncoder() class, which comes with sklearn.preprocessing. You can read the documentation of LabelEncoder [here](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html).

In previous models such as linear and logistic regression, we had created dummy variables for hot encoding to categorical variables, since those models (being mathematical equations) can process only numeric variables. There the algorithm teats the numeric variables as numbers. But in case of decision tree its just a label. The decision trees are not constraint by the use of numeric values alone. They can very well handle the categorical values. The reason we are using numeric labels just to distinguise one from the other. There is uniform underlying encoding we would have assign to each of such numbers. Beyond that we are not going to treat the number as a number here, means there is not going to be any arithmetic operation on these numeric labels.

# apply Label encoder to df\_categorical

le = preprocessing.LabelEncoder()

df\_categorical = df\_categorical.apply(le.fit\_transform)

﻿

### **Hyperparameter Tuning**

The default tree is quite complex, and we need to simplify it by tuning the hyperparameters. How do you control the complexity (or size) of a tree? A very ‘big’ or complex tree will result in overfitting. On the other hand, if you build a relatively small tree, it may not be able to achieve a good enough accuracy (i.e. it will underfit).

So what values of hyperparameters should you choose? As you would have guessed, you can use grid search cross-validation to find the optimal hyperparameters.

First, let's understand the parameters in a decision tree. You can read this in the documentation using help(DecisionTreeClassifier).

* **criterion** (Gini/IG or entropy): It defines the function to measure the quality of a split. Sklearn supports “gini” criteria for Gini Index & “entropy” for Information Gain. By default, it takes the value “gini”.
* **splitter**: It defines the strategy to choose the split at each node. Supports “best” value to choose the best split & “random” to choose the best random split. By default, it takes “best” value.
* **max\_features**: It defines the no. of features to consider when looking for the best split. We can input integer, float, string & None value.
  + If an integer is inputted then it considers that value as max features at each split.
  + If float value is taken then it shows the percentage of features at each split.
  + If “auto” or “sqrt” is taken then max\_features=sqrt(n\_features).
  + If “log2” is taken then max\_features= log2(n\_features).
  + If None, then max\_features=n\_features. By default, it takes “None” value.
* **max\_depth**: The max\_depth parameter denotes maximum depth of the tree. It can take any integer value or None. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min\_samples\_split samples. By default, it takes “None” value.
* **min\_samples\_split**: This tells above the minimum no. of samples reqd. to split an internal node. If an integer value is taken then consider min\_samples\_split as the minimum no. If float, then it shows percentage. By default, it takes “2” value.
  + min\_samples\_split is the minimum number of data points required in a node to be considered for further splitting.
* **min\_samples\_leaf**: The minimum number of samples required to be at a leaf node. If an integer value is taken then consider - -min\_samples\_leaf as the minimum no. If float, then it shows percentage. By default, it takes “1” value.
  + min\_samples\_leaf is the minimum number of samples required in a (resulting) leaf for the split to happen. Thus, if you specify a high value of min\_samples\_leaf, the tree will be forced to stop splitting quite early.
* **max\_leaf\_nodes**: It defines the maximum number of possible leaf nodes. If None then it takes an unlimited number of leaf nodes. By default, it takes “None” value.
* **min\_impurity\_split**: It defines the threshold for early stopping tree growth. A node will split if its impurity is above the threshold otherwise it is a leaf.

**What is the difference between min\_sample\_split and min\_sample\_leaf?**

min\_sample\_split  tells above the minimum no. of samples reqd. to split an internal node. If an integer value is taken then consider min\_samples\_split as the minimum no. If float, then it shows percentage. By default, it takes “2” value.

min\_sample\_leaf is the minimum number of samples required to be at a leaf node. If an integer value is taken then consider - -min\_samples\_leaf as the minimum no. If float, then it shows percentage. By default, it takes “1” value.

# **Comprehension - Hyperparameters**

Consider a decision tree classification model that has a very high training accuracy and a low test accuracy, i.e. the model has a high variance. The training accuracy is 98%, and the test accuracy is 55%. The ‘min\_samples\_split’ for this model is 5, and the ‘max\_depth’ is 20.

**Please note** that, unless explicitly specified, 'node' does not mean the terminal node (i.e. leaf) - it refers to an internal node.

**What does the min\_samples\_split = 5 imply? More than one option may be correct.**

**The minimum no. of samples required to split an internal node is equal to 5.**

**Feedback :**

The hyperparameter **min\_samples\_split** is the minimum no. of samples required to split an internal node. Its default value is 2, which means that even if a node is having 2 samples it can be further divided into leaf nodes.

**Correct**

**Even if a node is having 5 samples it can be further divided into leaf nodes.**

**Feedback :**

The hyperparameter **min\_samples\_split** is the minimum no. of samples required to split an internal node. Its default value is 2, which means that even if a node is having 2 samples it can be further divided into leaf nodes.

**Select all that apply. (More than one option may be correct.)**

**min\_samples\_split = 5 implies that the node should have at least five data points for splitting.**

**Feedback :** min\_samples\_split = 5 indicates that splitting will not be performed if the number of data points in the node is less than 5. The min\_samples\_split specifies the minimum number of data points a node should have for splitting to be attempted.

**The min\_samples\_split sets a lower bar on the number of data points a node should have.**

**Feedback :** The min\_samples\_split specifies the minimum number of data points a node should have for splitting to be attempted.

**Suppose you decide to tweak the hyperparameters so as to decrease the variance/overfitting. Which of the following steps will help? More than one option may be correct**.

**Increasing min\_samples\_split.**

**Feedback :** A low value of the min\_samples\_split will lead to a small number of data points in the nodes. This means that it is very likely that each leaf (obtained after splitting) is going to represent very few (or only one, in some cases) data points. So, you increase the min\_samples\_split.

Top of Form

**Decreasing max\_depth**

**Feedback :** Decreasing max\_depth will stop the tree to grow deeper, in that way your tree will not overfit the data and you will have a decent accuracy in both test and train.

**Correct**

Bottom of Form

**What will the (likely) impact of increasing the value of min\_sample\_splits from 5 to 10?**

**The depth will decrease.**

**Feedback :** *Since the node should now contain at least 10 data points before splitting, as opposed to 1, all the branches — where the nodes had less than 10 data points — will be chopped off, leading to a decrease in the tree depth.*

**Min\_sample\_leaf - What will be the effect on the depth of the tree if min\_sample\_leaf is set to 1? Will the tree be overfitting the train data or the test data?**

In the in-module question, my answer was

" The low or small value for min\_sample\_leaf the model tend to overfit.  min\_samples\_leaf is the minimum number of samples required in a (resulting) leaf for the split to happen. Thus, if you specify a high value of min\_samples\_leaf, the tree will be forced to stop splitting quite early.  It does not have any direct relation with tree depth.  "

The more the number of samples in a leaf node, lesser will be the depth of the tree and vice versa. If the min sample at a leaf is set to 1, the tree will be very deep.

# **What is the difference between node and a leaf node?**

Leaf node is the label, gives you the decision on certain attribute.

Node - is an internal node which can be further split and it is where the test is conducted.

# **Graded Questions**

**Training Set -** The first two columns in the training set file represent the two attributes, Attribute 1 (a1) and Attribute 2 (a2). The third column (column C) is the class label associated with each data point in the training set. There are two class labels: 1 and 2. In total, there are 30 data points/observations in the training set. You need to build a decision tree using this file. Use all 30 points for training. Construct the decision tree step-by-step as the following questions suggest.

**Validation Set** - The first two columns in the validation set file represent the two attributes, Attribute 1 (a1) and Attribute 2 (a2). The third column (column C) is the class label associated with each data point in the validation set. The fourth column is the class label as predicted by the decision tree model. There are two class labels: 1 and 2. In total, there are 4 data points/observations in the validation set. This data file will be needed to check the accuracy of pruned trees.

Please note that you mustn't use DecisionTreeClassifier or write codes to solve the following questions. You can use Excel to split and filter the data set according to the test conditions specified in the questions. The DecisionTreeClassifier function uses a slightly different test on each node, and hence, the final answer to the following questions may not match.

Please use the following hyperparameters to solve the following questions:

* max\_depth = 20
* min\_samples\_split = 10
* min\_samples\_leaf = 5
* Homogeneity measure = gini

Note: **(13, 5)**, written on a node, implies that 13 data points belong to class label 1, and 5 data points belong to class label 2.

**What is the Gini index of the data in the training set?**

(15/30)2+(15/30)2

**Feedback :***There are two class labels in the training data. So the Gini index will be equal to (probability of class label 1)^2 + (probability of class label 2)^2. The probability of class label 1 =*(1530)*, and the probability of class label 2 =*(1530)*.*

**What is the Gini index of the partitions if you split the training set on attribute1 (i.e. attribute1 < 7). (Note: You go left if attribute1 is less than 7, and you go right if attribute1 is greater than or equal to 7.)**

(0.5)∗((8/15)2+(7/15)2)+(0.5)∗((7/15)2+(8/15)2)

**Feedback :***The Gini index of the partition with a1 < 7 =*(8/15)2+(7/15)2*. The Gini index of the partition with a1 > =7 is*(7/15)2+(8/15)2*. The Gini index of all the partitions =*(15/30)*\*(Gini index of partition with a1 < 7) +*(15/30)*\*(Gini index of partition with a1 >= 7).*

**Which attribute will you split the training data on?**

Note: Find the Gini index for both cases and then decide the split.

**Attribute 2: (a2) < 20**

**Feedback :** *The Gini index(a2 < 20) = 0.65, and the Gini index(a1 < 7) = 0.50. Since Gini index(a2 < 20) > Gini index(a1 < 7), you split on attribute 2.*

|  |  |
| --- | --- |
| **Can you split the right node, with 2 data points belonging to class label 1 and 10 data points belonging to class label 2, on a1 = 2?** | **No. One of the leaves will have only 2 data points upon splitting, which is less than the min\_samples\_leaf (=5).**  **Feedback :**  *This split violates the 'minimum number of data points that a leaf should have on splitting' condition put up by the min\_samples\_leaf.* |

|  |  |
| --- | --- |
| **Let’s say that you further split the left partition on a1 = 2, 12. You get the following tree on splitting**  **What is the Gini index of the partitions obtained after splitting on a1 = 2,12? (Note: Choose the closest value.)** | **0.9**  **Feedback :**  *The Gini index of the partition with a1=2, 12 is*(1/6)2+(5/6)2*. The Gini index of the partition with a1!=2, 12 is*(12/12)2+(0/12)2*. The Gini index of all the partitions =*(6/18)*\*(Gini index of partition with a1=2,12) +*(12/18)*\*(Gini index of partition with a1!=2,12).*  **You cannot further split the left node. Why?**  (Note: The left node is the node highlighted in the following tree.)  The number of data points in this node is less than the minsplit.  **Feedback :***The number of data points in this node is 6. The value of the minsplit, as specified at the beginning, is 10.* |

# **Random Forests**

Random forests is one of the most popular algorithms in machine learning. Random forests use a technique known as **bagging**, which is an **ensemble** method. So before diving into random forests, let's first understand ensembles.

# **Ensembles (pronounce as awn·sawm·blz)**

An ensemble means a group of things viewed as a whole rather than individually. In ensembles, a **collection of models** is used to make predictions, rather than individual models. Arguably, the most popular in the family of ensemble models is the random forest: an ensemble made by the **combination of a large number of decision trees.**

In principle, ensembles can be made by combining all types of models. An ensemble can have a logistic regression, a neural network, and few decision trees working in unison.

Naturally, a few questions arise before you understand how ensembles work:

1. Why should a **collection of models work better**than individual ones?
2. How do you **choose the individual models** to form the ensemble, so that it is better than any of the individual models themselves?

# **Diversity and Acceptability**

Ensembles of models are somewhat analogous to teams of individual players. If you were to choose a football team, there would be two things you’d do:

1. Choose people with different skill sets, such as defenders, attackers, and a goalkeeper, to ensure **diversity**, and
2. Choose good players, i.e. ensure that all players are **acceptable** from a skill set point of view (and at least better than a regular person).

**Diversity** ensures that the models serve **complementary** purposes, which means that the individual models make predictions **independent of each other**. The advantages of this are different depending on the type of ensemble. In statistical terms diversity means Independence.

* 1. Answers of different models are not co-related
  2. Agreement between answers of different model is coincidental

For example, a random forest is an ensemble with a large number of trees as individual models. Diversity ensures that even if some trees overfit, the other trees in the ensemble will neutralize the effect. The independence among the trees results in a **lower variance** of the ensemble compared to a single tree. We will soon discuss how the learning algorithm is designed to achieve independence, and how this is beneficial.

If two models give the same answers on a random data, it will be totally coincidental

**Feedback :**Diversity represents independence, i.e. models are not correlated (and do not get influenced by) other models. This means that the answers (predictions) given by two models are independent of each other (you’ll study how this is achieved in a short while).

**Acceptability** implies that each model is at least **better than a random model**. This is a pretty lenient criterion for each model to be accepted into the ensemble, i.e. it has to be at least better than a random guesser. If we consider a binary classification task, then better than a random model implies that the model under consideration   
Makes correct predictions with a probability statistically better than that of a random model, i.e. 0.5

**Feedback :** Acceptability means that a model is at least not making random guesses, whose P(success) is 0.50. Thus, we want models whose probability of success is > 50%.

But how can you guarantee that if you fulfil these two conditions to make an ensemble, it will be better than any individual model?

Now, to understand how an ensemble makes decisions, consider an ensemble with 100 models comprising of decision trees, logistic regression models, etc. Given a new data point, each model will predict an output y for this data point. If this is **binary classification**, then you simply take the majority score. If more than 50% models say y = 0, you go with 0 and vice-versa. This means that if there are n models in the ensemble, and more than half of them give you the right answers, you will make the right decision; if more than n/2 are wrong, you will make a wrong decision. In the **coin toss analogy**, making a **correct prediction** corresponds to **head,** whereas getting a **wrong prediction** corresponds to **tail**.

If we can prove that the probability of more than half the models making a wrong prediction is lesser than that of any of the individual models, we know that the ensemble is a better choice than any of the individual models.

The question is, why should you expect the majority vote to perform better on unseen data than any of the individual 100 models? There are a number of convincing arguments to answer this.

Firstly, if each of the individual models is **acceptable**, i.e. they’re wrong with a probability less than 50%, you can show that the probability of the ensemble being wrong (i.e. the majority vote going wrong) will be far lesser than that of any individual model.

Also, the ensembles avoid getting misled by the **assumptions made by individual models**. For example, ensembles (particularly random forests) successfully reduce the problem of overfitting. If a decision tree in an ensemble overfits, you let it. Chances are extremely low that more than 50% of the models have overfitted. Ensembles make sure that you do not put all your eggs in one basket.

Remember that each model in an ensemble is **acceptable**, i.e. the probability of each model being wrong is less than 0.5 (since a random binary classification model is right 50% of the time).

Using an analysis, you can show that the probability that more than half of the models in an ensemble make the wrong prediction, is **significantly less than 0.5**, i.e. a random model.

You can use the (biased) coin toss analogy to do this. **Map the predictions** made by an ensemble to the **two sides of a biased coin**. Getting a prediction **correct** is equivalent to getting **heads**, and getting it **wrong** is equivalent to **tails**, i.e. you map heads to success (correct prediction) and tails to an incorrect prediction.

**Coin Toss Analogy**

Which of the following assumptions make it possible to use the coin toss analogy? Mark all that apply

Each toss is independent of each other and so are the models in an ensemble

**Feedback :**We have assumed that 1) the models are independent and 2) they are all individually acceptable. The coin, with success mapped to heads, is biased towards heads. Thus, P(heads > 0.5) and since all models are acceptable, P(success) > 0.5.

**Correct**



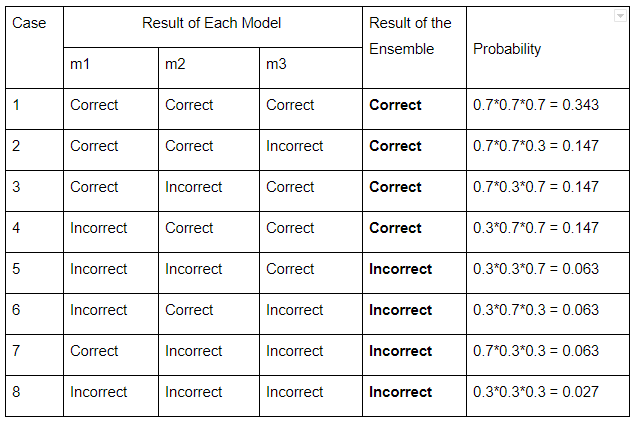
The coin is biased towards and favors heads, and each model is acceptable

**Experimenting with an Ensemble of Three Models**

To understand why ensembles work better than individual models, let’s take a simple example with three coins (models). Consider an ensemble of **three models, m1, m2 and m3,** for a **binary classification** task (say, 1 or 0). Suppose each of these models has a **probability of being correct 70% of the time.**

So each model is acceptable. Given a data point whose class has to be predicted, the ensemble will predict the class using a **majority score**. In other words, if two or more models predict class = 1 as the output, the ensemble will predict 1 and vice versa.

The following table shows all the possible cases that can occur while classifying a test data point as 1 or 0. The column to the extreme right shows the probability of each case.



**Ensemble Model**

In the table, there are 4 cases each where the decision of the final model (ensemble) is either correct or wrong. Let’s assume that the probability of the ensemble being correct is p, and the probability of the ensemble being wrong is q.

For the data in the table, p and q can be calculated as follows:

p = 0.343 + 0.147 + 0.147 + 0.147 = 0.784

q = 0.027 + 0.063 + 0.063 + 0.063 = 0.216 = 1 - p

Notice how the ensemble has a higher probability of being correct and a lower probability of being wrong than any of the individual models (0.78 > 0.700 and 0.216 < 0.300). In this way, you can also calculate the probabilities of the ensemble being correct and incorrect with 4, 5, 100, 1000, and even a million individual models. The difference in probabilities will increase with an increasing number of models, thus **improving the overall performance of the ensemble**.

We call **weak learners** (or **base models**) models that can be used as building blocks for designing more complex models by combining several of them. Most of the time, these basics models perform not so well by themselves either because they have a high bias (low degree of freedom models, for example) or because they have too much variance to be robust (high degree of freedom models, for example). Then, the idea of ensemble methods is to try reducing bias and/or variance of such weak learners by combining several of them together in order to create a **strong learner** (or **ensemble model**) that achieves better performances.

# Combine weak learners

In order to set up an ensemble learning method, we first need to select our base models to be aggregated. Most of the time (including in the well known bagging and boosting methods) a single base learning algorithm is used so that we have homogeneous weak learners that are trained in different ways. The ensemble model we obtain is then said to be “homogeneous”. However, there also exist some methods that use different type of base learning algorithms: some heterogeneous weak learners are then combined into an “heterogeneous ensembles model”.

One important point is that our choice of weak learners should be **coherent with the way we aggregate these models**. If we choose base models with low bias but high variance, it should be with an aggregating method that tends to reduce variance whereas if we choose base models with low variance but high bias, it should be with an aggregating method that tends to reduce bias.

This brings us to the question of how to combine these models. We can mention three major kinds of meta-algorithms that aims at combining weak learners:

* **bagging**, that often considers homogeneous weak learners, learns them independently from each other in parallel and combines them following some kind of deterministic averaging process
* **boosting**, that often considers homogeneous weak learners, learns them sequentially in a very adaptative way (a base model depends on the previous ones) and combines them following a deterministic strategy
* **stacking**, that often considers heterogeneous weak learners, learns them in parallel and combines them by training a meta-model to output a prediction based on the different weak models predictions

Very roughly, we can say that bagging will mainly focus at getting an ensemble model with less variance than its components whereas boosting and stacking will mainly try to produce strong models less biased than their components (even if variance can also be reduced).

# Bootstrapping

Let’s begin by defining bootstrapping. This statistical technique consists in generating samples of size B (called bootstrap samples) from an initial dataset of size N by randomly drawing with replacement B observations.

|  |  |
| --- | --- |
| https://miro.medium.com/max/1575/1*lWnm3eJVe3uo95OcSg5jUA@2x.png  Under some assumptions, these samples have pretty **good statistical properties**: in first approximation, they can be seen as being drawn both directly from the true underlying (and often unknown) data distribution and independently from each others. | So, they can be considered as representative and independent samples of the true data distribution (almost i.i.d. samples). The hypothesis that have to be verified to make this approximation valid are two fold. First, the size N of the initial dataset should be large enough to capture most of the complexity of the underlying distribution so that sampling from the dataset is a good approximation of sampling from the real distribution (**representativity**). Second, the size N of the dataset should be large enough compared to the size B of the bootstrap samples so that samples are not too much correlated (**independence**). Notice that in the following, we will sometimes make reference to these properties (representativity and independence) of bootstrap samples: the reader should always keep in mind that **this is only an approximation**. |

Bootstrap samples are often used, for example, to evaluate variance or confidence intervals of a statistical estimators. By definition, a statistical estimator is a function of some observations and, so, a random variable with variance coming from these observations. In order to estimate the variance of such an estimator, we need to evaluate it on several independent samples drawn from the distribution of interest. In most of the cases, considering truly independent samples would require too much data compared to the amount really available. We can then use bootstrapping to generate several bootstrap samples that can be considered as being “almost-representative” and “almost-independent” (almost i.i.d. samples). These bootstrap samples will allow us to approximate the variance of the estimator, by evaluating its value for each of them.

The selection of all the example in the dataset has equal probability. This method can help to better understand the mean and standand deviation from the dataset.

Let’s assume we have a sample of ‘n’ values (x) and we’d like to get an estimate of the mean of the sample.

mean(x) = 1/n \* sum(x)

We know that our sample is small and that our mean has error in it. We can improve the estimate of our mean using the bootstrap procedure:

1. Create many (e.g. m) random sub-samples of our dataset with replacement (meaning we can select the same value multiple times).
2. Calculate the mean of each sub-sample.
3. Calculate the average of all of our collected means and use that as our estimated mean for the data.

For example, let’s say we used 3 resamples and got the mean values 2.5, 3.3 and 4.7. Taking the average of these we could take the estimated mean of the data to be 3.5.

One of the many reasons bootstrapping has become common is because of the increase in computing power. This allows for many more permutations to be done with different resamples than otherwise possible. Bootstrapping is used in both bagging and boosting

# Bagging

When training a model, no matter if we are dealing with a classification or a regression problem, we obtain a function that takes an input, returns an output and that is defined with respect to the training dataset. Due to the theoretical variance of the training dataset (we remind that a dataset is an observed sample coming from a true unknown underlying distribution), the fitted model is also subject to variability: **if another dataset had been observed, we would have obtained a different model**.

The idea of bagging is then simple: we want to fit several independent models and “average” their predictions in order to obtain a model with a lower variance. However, we can’t, in practice, fit fully independent models because it would require too much data. So, we rely on the good “approximate properties” of bootstrap samples (representativity and independence) to fit models that are almost independent.

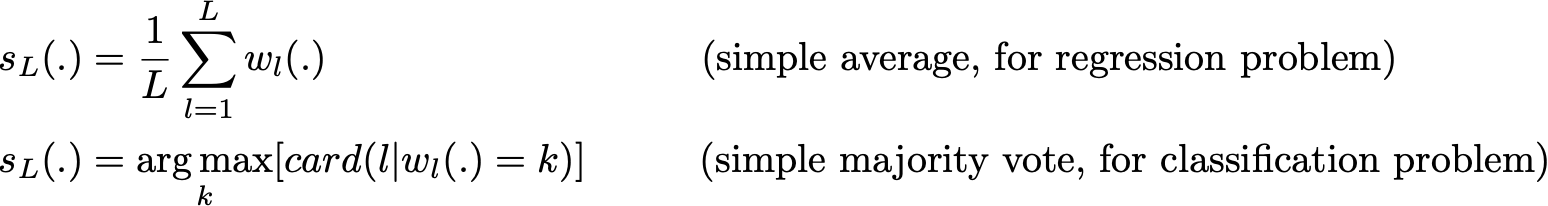
First, we create multiple bootstrap samples so that each new bootstrap sample will act as another (almost) independent dataset drawn from true distribution. Then, we can **fit a weak learner for each of these samples and finally aggregate them such that we kind of “average” their outputs** and, so, obtain an ensemble model with less variance that its components. Roughly speaking, as the bootstrap samples are approximatively independent and identically distributed (i.i.d.), so are the learned base models. Then, “averaging” weak learners outputs do not change the expected answer but reduce its variance (just like averaging i.i.d. random variables preserve expected value but reduce variance).

So, assuming that we have L bootstrap samples (approximations of L independent datasets) of size B denoted

https://miro.medium.com/max/2288/1*nu96mPOtrXosJYgWA4Rvbw@2x.png

we can fit L almost independent weak learners (one on each dataset) https://miro.medium.com/max/443/1*Dn6v09t5_L5cvADxJHJzHQ@2x.png

and then aggregate them into some kind of averaging process in order to get an ensemble model with a lower variance. For example, we can define our strong model such that



There are several possible ways to aggregate the multiple models fitted in parallel. For a regression problem, the outputs of individual models can literally be averaged to obtain the output of the ensemble model. For classification problem the class outputted by each model can be seen as a vote and the class that receives the majority of the votes is returned by the ensemble model (this is called **hard-voting**). Still for a classification problem, we can also consider the probabilities of each classes returned by all the models, average these probabilities and keep the class with the highest average probability (this is called **soft-voting**). Averages or votes can either be simple or weighted if any relevant weights can be used.

Finally, we can mention that one of the big advantages of bagging is that **it can be parallelized**. As the different models are fitted independently from each others, intensive parallelization techniques can be used if required.

Further clarifying the steps of bagging includes

1. Suppose there are N observations and M features. A sample from observation is selected randomly with replacement(Bootstrapping).
2. A subset of features are selected to create a model with sample of observations and subset of features.
3. Feature from the subset is selected which gives the best split on the training data.
4. This is repeated to create many models and every model trained in parallel
5. Prediction is given based on the aggregation of predictions from all the models.

# Random forests

Learning trees are very popular base models for ensemble methods. Strong learners composed of multiple trees can be called “forests”. Trees that compose a forest can be chosen to be either shallow (few depths) or deep (lot of depths, if not fully grown). Shallow trees have less variance but higher bias and then will be better choice for sequential methods that we will described thereafter. Deep trees, on the other side, have low bias but high variance and, so, are relevant choices for bagging method that is mainly focused at reducing variance.

The **random forest** approach is a bagging method where **deep trees**, fitted on bootstrap samples, are combined to produce an output with lower variance. However, random forests also use another trick to make the multiple fitted trees a bit less correlated with each others: when growing each tree, instead of only sampling over the observations in the dataset to generate a bootstrap sample, we also **sample over features** and keep only a random subset of them to build the tree.

Sampling over features has indeed the effect that all trees do not look at the exact same information to make their decisions and, so, it reduces the correlation between the different returned outputs. Another advantage of sampling over the features is that**it makes the decision making process more robust to missing data**: observations (from the training dataset or not) with missing data can still be regressed or classified based on the trees that take into account only features where data are not missing. Thus, random forest algorithm combines the concepts of bagging and random feature subspace selection to create more robust models.

# Random Forest vs Bagging

To understand random forests, you will first need to understand **bagging**, an ensemble method. Bagging stands for **bootstrapped aggregation.**It is a technique for choosing random samples of observations from a dataset. Each of these samples is then used to train each tree in the forest.

As you study bagging, keep in mind that it is just a **sampling technique** and is **not specific to random forests.** In the bagging type of ensembles, random forests are by far the most successful. They are essentially ensembles of a **number of decision trees**. You create a large number of models (say, 100 decision trees), each one on a **different bootstrap sample**, from the training set. To get the result, you **aggregate** the decisions taken by all the trees in the ensemble.

**Bootstrapping** means creating bootstrap samples from a given data set. A bootstrap sample is created by sampling the given data set **uniformly** and **with replacement**. A bootstrap sample typically contains about 30-70% data from the data set. **Aggregation** implies combining the results of different models present in the ensemble.

|  |  |
| --- | --- |
|  | In every split at a node it will pick a random subset of the attributes and explores only those to pick the one which gives maximum increase in homogeneity.  The additional twist here is also picking a random sample of features everytime I’m attempting a split during the process of the construction of decision tree. This is very significant. |

Consider the heart disease data set where a few attributes such as Thal, blood pressure, etc., are prominent predictors for the target variable. If you were to build multiple decision trees on this as a part of an ensemble, considering all the attributes for all the individual trees, which of these violations would occur and be significant?

This model will not be diverse enough which breaks one of the primary characteristics in ensemble methodology. If a few variables are prominent, a large number of trees will have them as important nodes. Because whenever we bootstrap samples these prominent features will always sit on the top of the tree. So the trees would look similar. Similar trees violate the condition of diversity. So to **counter that we need to make sure every time we attempt a split we are actually considering a random sample of features, not all the features**.

* Random selection of features for splitting
* Not selecting all the features
* This way of random selection of features for splitting, it is quite possible that for many of the nodes we may not consider most prominent features

Suppose you want to build a random forest of 10 decision trees. First, you will create 10 bootstrap samples from the data and then, **train** each tree on a **different bootstrap sample**. Finally, while predicting a test case, each tree will make a prediction, and the final prediction will be the **majority score** of all these predictions.

**When do observation and feature sampling take place for trees inside a random forest? More than one option may be correct**.

A random subset of observations is chosen every time a new tree is built in a forest.

**Feedback :**A different random subset of observations is chosen, which is called the bootstrap sample, for each tree that is to be built in the forest. This is called bootstrapping. **Correct**



A random subset of features is chosen every time a node is being split inside a tree.

**Feedback :**After the bootstrap sample is selected, tree building starts, and a random subset of features is selected at each node in order to split it. This is what makes random forests even better than a simple bagging algorithm.

**How is a random forest different from bagging?**

In a random forest, a random sample of features is chosen at each node split, which does not happen in bagging.

**Feedback :**

Bagging includes the creation of different bootstrap samples for different models, and aggregating the results of the models. Random forests use this technique along with randomly selecting features at each node while splitting it.

**During bagging, or bootstrap aggregation, the test data point is passed through all the trees of the forest, and each tree makes its own prediction. How are these predictions aggregated in the case of a regression problem?**

The final prediction is the mean of all the predictions of the individual decision trees.

# Advantages of Random Forests

**Diversity** arises because you create each tree with a **subset of the attributes/features/variables**, i.e. you don’t consider all the attributes while making each tree. The choice of the attributes considered for each tree is **random**. This ensures that the trees are independent of each other.

**Stability arises** because the answers given by a large number of trees average out. A random forest has a **lower model variance** than an ordinary individual tree.

**Immunity to the curse of dimensionality**: Since each tree does not consider all the features, the feature space (the number of features a model has to consider) reduces. This makes the algorithm immune to the curse of dimensionality. A large feature space causes computational and complexity issues.

**Parallelizability**: You need a number of trees to make a forest. Since two trees are independently built on different data and attributes, they can be built separately. This implies that you can make full use of your multi-core CPU to build random forests. Suppose there are 4 cores and 100 trees to be built; each core can build 25 trees to make a forest.

**Testing /training** data and the **OOB or out-of-bag error**: You always want to avoid violating the fundamental tenet of learning: “not testing a model on what it has been trained on”. While building individual trees, you choose a **random** subset of the observations to train it. If you have 10,000 observations, each tree may only be made from 7000 (70%) randomly chosen observations. OOB is the mean prediction error on each training sample xᵢ, using only the trees that do not have xᵢ in their bootstrap sample. If you think about it, this is very similar to a **cross-validation error**. In a CV error, you can measure the performance on the subset of data the model hasn’t seen before.

In fact, it has been proven that using an OOB estimate is as accurate as using a test data set of a size equal to the training set.

Thus, the OOB error completely omits the need for set-aside test data (though you can still work with test data like you have been doing, at the cost of eating into the training data).

|  |  |
| --- | --- |
|  | Essentially, we are checking the performance of each tree for a data set which it didn’t see. There are multiple trees and all are built from different bootstrap samples.  So the OOB error is treated as test performance of the random forest algorithm. |

**The core idea behind bagging is that of a majority score rather than committing to set of assumptions made by a single model. The idea is particularly successful in random forests because trees**:

Are typically unstable

**Feedback :***If you have only one tree, you have to rely on the decision it makes. The decision a single tree makes (on unseen data) depend highly on the training data since trees are unstable. In a forest, even if a few trees are unstable, averaging out their decisions ensures that you are not making mistakes because of a few trees’ unstable behaviour.*

**Random Forests vs Decision Trees -** In terms of accuracy, is a random forest always better than a decision tree?

False

**Feedback :***While it is well known that random forests are better, in terms of accuracy, than a single decision tree, it cannot be said that they are better than every possible decision tree. It is just more difficult to build a decision tree that is better than a random forest. In fact, there may be several trees that provide better predictions on unseen data.*

**Variance in Random Forests -** Which of the following statements is true?

A larger number of trees will result in a lower variance of the ensemble.

**Feedback :**

*Variance means how much a model (ensemble here) changes with changes in the training data. If a large number of trees is at work, then even if some of them show a high instability (extreme variation in the trees and their predictions), the ensemble as a whole will reduce the variance by averaging out the results of each tree.*

# Comprehension - OOB (Out-of-Bag) Error

In the last segment, you learnt that the **OOB error** is almost as good as the **cross-validation error**. The final prediction is the **aggregation** of all the predictions of individual decision trees. Remember that each tree in a random forest is only trained on a **random subset** of the training set, which is called a **bootstrapped sample**. This means that for each sample (observation), there are several trees that did not include that sample, and to these trees, this sample is **unseen**. Let’s understand this better.

Suppose there are N **=** **100 observations** with M **= 15 features**, and the outcome variable is a categorical variable Y. Also, you build a random forest with **50 trees**. The OOB is calculated as follows:

For each observation Ni, Ni is passed to all the trees that did not have it in their training. These trees then predict the class of Ni. The final prediction about Ni is decided by a majority vote.

Now let’s apply this to N1. Suppose 10 trees did not have N1 in their training. But these 10 trees make their prediction about N1. Let’s say 4 trees predicted 0, and the other 6 predicted 1, as the output. The final prediction about N1 will be 1.

Next, we move on to N2. Suppose 15 trees did not have N2 in their training. But these 15 trees make their prediction about N2. Let’s say 12 predicted 0, and the rest 3 predicted 1. The final prediction about N2 will be 0.

This is done for each observation in the training set. Once all the predictions for each observation are calculated, the OOB error is calculated as the number of observations predicted wrongly as a proportion of the total number of observations.

**Which of the following dataset is used to calculate the OOB error?**

Training set

**Feedback :** *Only the training set is used while calculating the OOB error, which is why it gives a good idea of model performance on the unseen data without using a test set.*

All the observations of the training set are used to calculate the OOB error.

# Comprehension - Time Taken to Build a Random Forest

Consider a dataset with M features and n observations. Each of the n observations is thus represented as the vector Xj={xj1,xj2,xj3,...,xjM} having the output Yi.

There are two critical steps in building a random forest:

1. Bagging, or choosing observations in a bootstrapped manner and creating individual trees
2. Random subsetting of features while creating each tree and finding the right splits

If we decide to have S trees, each of the Si tree will use a bootstrapped sample of observations to train themselves. Note that the sampling is done with replacement, i.e. there will be observations common among trees, but there will not be two trees which have been trained on exactly the same set of observations.

While creating each tree, a random subset of features will be considered for splitting at each node. A typical way is to consider √M number of features at each split.

To make a prediction on a data point Xj={xj1,xj2,xj3,...,xjM}, the forest uses all the Si trees, calculates the outputs from each tree and takes the majority vote.

**Time Taken to Build an Ensemble**

We can estimate the computational time required for building ensembles and making predictions.

Building an ensemble requires the following steps:

1) Take a random sample of observations, say j= 40% of the total and

2) Build S trees by finding all the splits within each tree from a subsetted feature space. The time required is thus proportional to the number of trees S and the time required to build each tree.

The time required to build each tree is proportional to the average number of levels of a tree and the average time required to find the split at each level. To find the right split, only a subset of features f is considered.

Further, the expected number of levels in a tree is given by log(n) where n is the number of observations being considered to build the tree

If there are S trees in a forest, M features (income, age etc.) and n observations (in the original training data), the time taken to build the forest depends on S, M and n. The time required will obviously depend on S. While building each of the S trees, time is spent in creating the levels of trees and time required to find splits among f features. **Levels of trees are given by log(n)**. Finding the right split depends on both n observations and f features because homogeneity will be measured for all f features and n observations.

**Time Spent on Splitting**

Consider building a single individual tree in an ensemble by taking j = 40% observations randomly from the training set. There are M features and n observations in the original training data. The **time spent at each split**in this tree is proportional to => sqrt(M).n.j ~= sqrt(M)\*n\*log(n)

Each split is made by comparing the homogeneity across j= 40% of the n observations. Thus, it has to depend on j and n (more the observations, more the time required to compare homogeneity). The time required to find a split also depends upon the number of features being considered which is sqrt(M).

# **Python sklearn RandomForestClassifier library**﻿

The following hyperparameters are present in a random forest classifier. Note that most of these hypereparameters are actually of the decision trees that are in the forest.

* **n\_estimators**: integer, optional (default=10): The number of trees in the forest.
* **criterion**: string, optional (default=”gini”)The function to measure the quality of a split. Supported criteria are “gini” for the Gini impurity and “entropy” for the information gain. Note: this parameter is tree-specific.
* **max\_features** : int, float, string or None, optional (default=”auto”)The number of features to consider when looking for the best split:
  + If int, then consider max\_features features at each split.
  + If float, then max\_features is a percentage and int(max\_features \* n\_features) features are considered at each split.
  + If “auto”, then max\_features=sqrt(n\_features).
  + If “sqrt”, then max\_features=sqrt(n\_features) (same as “auto”).
  + If “log2”, then max\_features=log2(n\_features).
  + If None, then max\_features=n\_features.
  + Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than max\_features features.
* **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 an internal node:\*\*
  + \*\*If int, then consider min\_samples\_split as the minimum number.
  + \*\*If float, then min\_samples\_split is a percentage and ceil(min\_samples\_split, n\_samples) are the minimum number of samples for each split.
* **min\_samples\_leaf** : int, float, optional (default=1)The minimum number of samples required to be at a leaf node:\*\*
  + **If int, then consider min\_samples\_leaf as the minimum number.**
  + **If float, then min\_samples\_leaf is a percentage and ceil(min\_samples\_leaf \* n\_samples) are the minimum number of samples for each node.**
* **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.

Recall that random forest takes a **sample of features** while splitting each node of its component trees. You can control the number of features considered at each split by specifying the **max\_features**hyperparameter. Let's now understand why max\_features is such an important attribute while building a random forest. The idea of picking a random sample of features to consider into the split at every stage of the algorithm is a crucial step to ensuring that the trees we get as part of the ensemble are actually different from each other significantly. There is lot of variability among the trees which makes the entire collection of ensemble a very robust collection.

For example, the hyperparameter max\_features specifies the maximum number of features considered at each node's split. For example, if the dataset contains 25 features, and you specify max\_features=4, then at each split (in the component trees), only a maximum of 4 randomly chosen features will be considered. How do you think the ensemble performance would vary as you gradually increase the value of max\_features?

**Ans:** At very low values of max\_features (e.g. 2), both the training and test accuracies will be low; both accuracies will gradually increase with max\_features up to a certain point, while at extremely high values (e.g. 19), the training accuracy will continue to increase while the test accuracy will reduce (and thus the model will overfit)

**The Effect of max\_features**

You saw that there is an optimal value of max\_features - at very **low values**, the component trees are**too simple to learn anything useful**, while at extremely high values, the component trees **become similar to each other**(and violate the 'diversity' criterion).

**The Effect of n\_estimators**

Also, when you observe the plot of n\_estimators and training and test accuracies, you will see that the as you increase the value of n\_estimators, both the training test accuracies gradually increase. More importantly, the model does *not*overfit even when its complexity is increasing. This is an important benefit of random forests - you can increase the number of trees as much you like without worrying about overfitting (if your computational resources allow).

We can now find the optimal hyperparameters using GridSearchCV.

|  |  |
| --- | --- |
| # Create the parameter grid based on the results of random search  param\_grid = {  'max\_depth': [4,8,10],  'min\_samples\_leaf': range(100, 400, 200),  'min\_samples\_split': range(200, 500, 200),  'n\_estimators': [100,200, 300],  'max\_features': [5, 10]  }  # 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)  # Fit the grid search to the data  grid\_search.fit(X\_train, y\_train)  **out:**  Fitting 3 folds for each of 72 candidates, totalling 216 fits  [Parallel(n\_jobs=-1)]: Done 42 tasks | elapsed: 56.8s  [Parallel(n\_jobs=-1)]: Done 192 tasks | elapsed: 6.8min  [Parallel(n\_jobs=-1)]: Done 216 out of 216 | elapsed: 8.1min finished  GridSearchCV(cv=3, error\_score='raise',  estimator=RandomForestClassifier(bootstrap=True, class\_weight=None, criterion='gini', max\_depth=None, max\_features='auto', max\_leaf\_nodes=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=10, n\_jobs=1, oob\_score=False, random\_state=None, verbose=0, warm\_start=False),  fit\_params=None, iid=True, n\_jobs=-1,  param\_grid={'max\_features': [5, 10], 'n\_estimators': [100, 200, 300], 'max\_depth': [4, 8, 10], 'min\_samples\_split': range(200, 500, 200), 'min\_samples\_leaf': range(100, 400, 200)},  pre\_dispatch='2\*n\_jobs', refit=True, return\_train\_score='warn', scoring=None, verbose=1) | # printing the optimal accuracy score and hyperparameters  print('We can get accuracy of',grid\_search.best\_score\_,'using',grid\_search.best\_params\_)  We can get accuracy of 0.818285714286 using {'max\_features': 10, 'n\_estimators': 200, 'max\_depth': 8, 'min\_samples\_split': 200, 'min\_samples\_leaf': 100}  ------------------------------------------------------------------------  # model with the best hyperparameters  from sklearn.ensemble import RandomForestClassifier  rfc = RandomForestClassifier(bootstrap=True,  max\_depth=10,  min\_samples\_leaf=100,  min\_samples\_split=200,  max\_features=10,  n\_estimators=100)  #fit  rfc.fit(X\_train,y\_train)  #predict  Predictions = rfc.predict(X\_test)  # evaluation metrics  from sklearn.metrics import classification\_report,confusion\_matrix  print(classification\_report(y\_test,predictions))  precision recall f1-score support  0 0.84 0.96 0.90 7058  1 0.70 0.36 0.47 1942  avg / total 0.81 0.83 0.81 9000  print(confusion\_matrix(y\_test,predictions))  [[6756 302]  [1249 693]] |