Lecture Notes

Decision Trees

Decision Trees naturally represent the way we make decisions. Think of a machine learning model as a decision-making engine that takes a decision on any given input object (data point). Imagine a doctor making a decision (the diagnosis) on whether a patient is suffering from a particular condition given the patient data, an insurance company making a decision on whether claims on a particular insurance policy needs to be paid out or not given the policy and the claim data, a company deciding on which role an applicant seeking a position in the company is eligible to apply for, based on the past track record and other details of the applicant, etc.. Solutions to each of these can be thought of as machine learning models trying to mimic the human decision making.

Refer to Figures 1 and 2 for a couple of examples built from representative UCI datasets. The Bank Marketing dataset (Figure 1) consists of data "is related with direct marketing campaigns of a Portuguese banking institution. The marketing campaigns were based on phone calls. Often, more than one contact to the same client was required, in order to access if the product (bank term deposit) would be ('yes') or not ('no') subscribed." The Heart dataset Figure 2) consists of data about various cardiac parameters along with an indicator column that says whether the person has a heart disease or not.

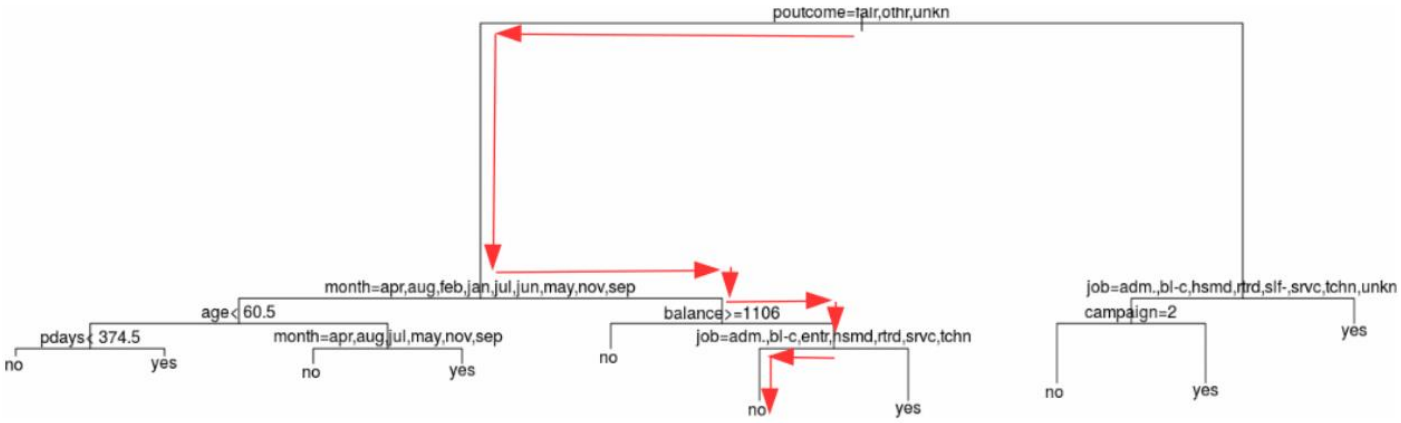


Figure: 1

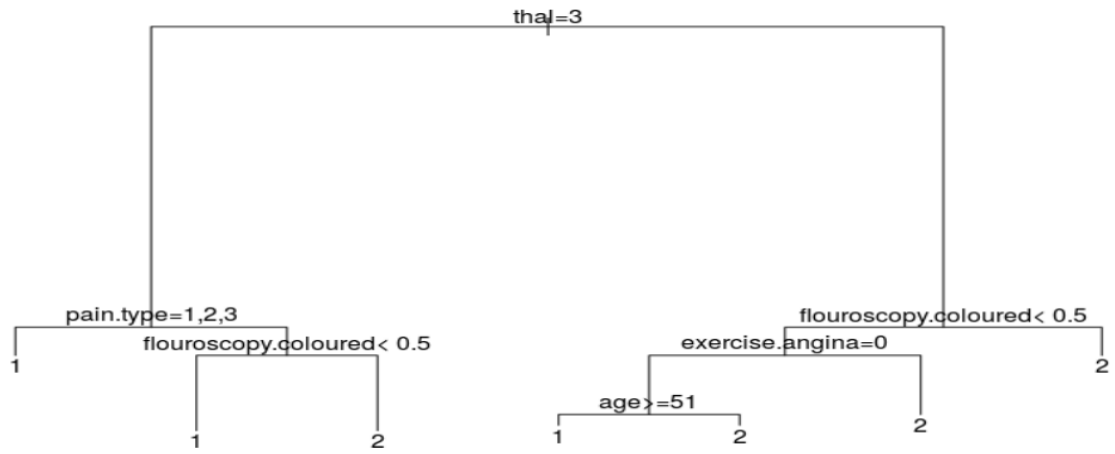


Figure: 2

Introduction

Without getting into the domain details of each of the terms in the datasets in fact the decision trees can be interpreted quite naturally. In the Bank Marketing example shown in Figure 1 the leaf nodes (bottom) are labelled yes (the customer will subscribe for a term deposit) or no (the customer will not subscribe for a term deposit).

The decision tree predicts that if the outcome of the call with the customer was fail, the contact month was march, if the current balance exceeds $1106 but the customer is unemployed then he/she will not subscribe to a term deposit — the path indicated by the red arrows in Figure 1. Note that every node (split junction) in the tree represents a test on some attribute of the data. As a matter of convention we go to the left part of the tree if the test passes, else go the right subtree.

The example given above represents the path left->right->right->left starting from the top (the root). For the heart dataset the leaf nodes (bottom) are labelled 1 (no heart disease) or 2 (has heart disease). The decision tree model predicts that if a person has thal of type 3 (normal), pain.type other than {1,2,3} and the number of blood vessels flouroscopy.coloured more than 0.5, then the person has heart disease. The example given above represents the path left->right->right starting from the top (the root). In general in a decision tree:

- The leaf nodes represent the final decisions.
- Each intermediate node represents a simple test on one of the attributes.
- The path from the root to a leaf corresponds to a conjunction of tests at each of the nodes
- on the path. We say a test data point 'follows a path' on a decision tree if it passes all the tests on the path in the decision tree. The branches out of an intermediate node are exclusive — the test data point can follow exactly one branch out of every intermediate node it encounters.
- The prediction by a decision tree on a data point is the one corresponding to the leaf at which the path followed by the data point ends.
- There could be multiple leaves representing the same class (decision). For example in the heart disease example, this simply means that a person does not have heart disease if: (thal=3 and pain.type in {1,2,3}) or (thal=3 and pain.type not in {1,2,3} and flouroscopy.coloured<0.5) or (thal!=3 and flouroscopy.coloured<0.5 and exercise.angina=0 and age>=51). Note that when the thal is normal, by and large the heart is normal.
- So in some sense the thal type is a major indicator of heart disease (this is apparent from the length of the leader lines from the root node). The last condition (1 leaf on the right branch) may seem a little counter-intuitive. An abnormal thal (right branch) is probably expected at age beyond 51 and so is not considered heart diseased, whereas at an age below 51 would be considered heart disease. In general the decision by tree is a value y represented by some of the leaves if the OR of the conditions corresponding to the paths from the root to each of the leaves with value y, is true for the given data point.

We generally assume, at least for explanation, the decision trees we consider are binary — every intermediate node has exactly two children. This is not a restriction since any more general tree can be converted into an equivalent binary tree. In practice however splits on attributes that have too many distinct values (for example a continuous valued attribute) are usually implemented as binary splits and splits on attributes with not many distinct values are implemented as multi-way splits. Figure 3 illustrates multiway split on an attribute A.

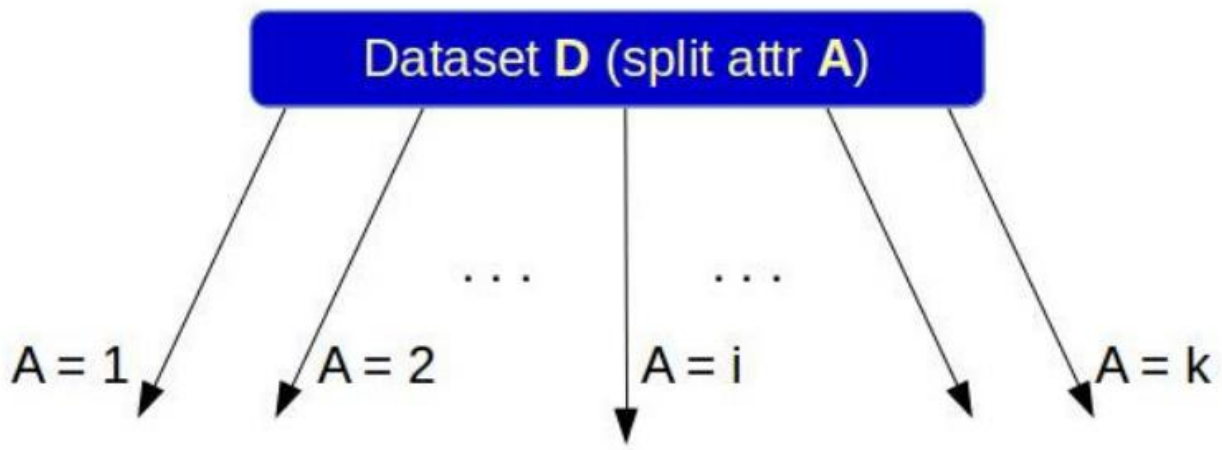


Figure 3: Multiway Split

The examples we have given are of the binary classification kind. However it is easy to see that this extends to multiclass classification as well with any change whatsoever to our description of a decision tree given above — the leaves would simply represent various class labels. It is also possible to extend decision trees to regression. Consider the dataset shown in Figure 4. It is a simple synthetic dataset where the y-value is just a constant with some noise thrown in three ranges of x-values — $0 < x \leq 1000$, $1000 < x \leq 2000$ and $2000 < x \leq 3000$. The decision tree identifies these three ranges and assigns the average y-value to each range.

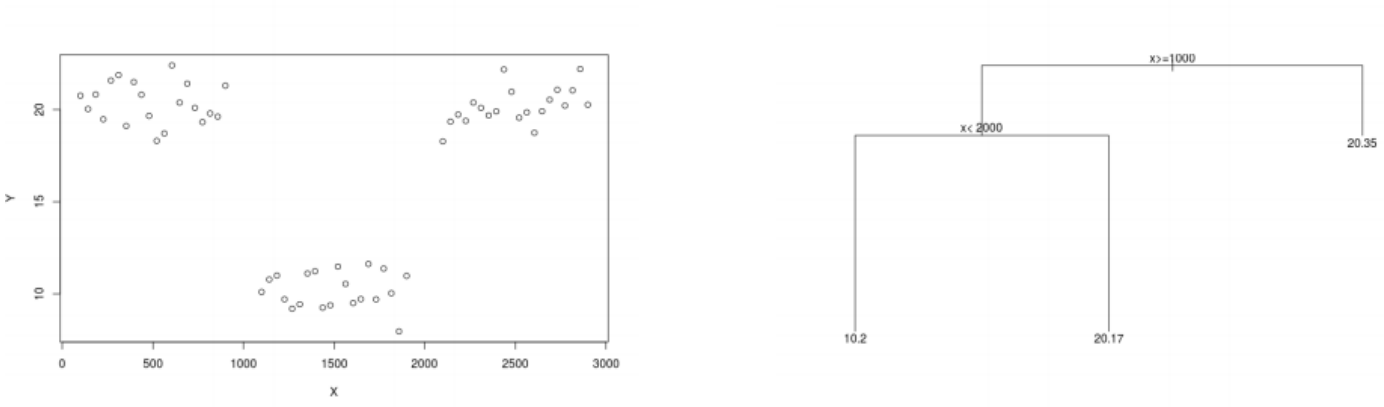


Figure 4: Decision Tree Regression

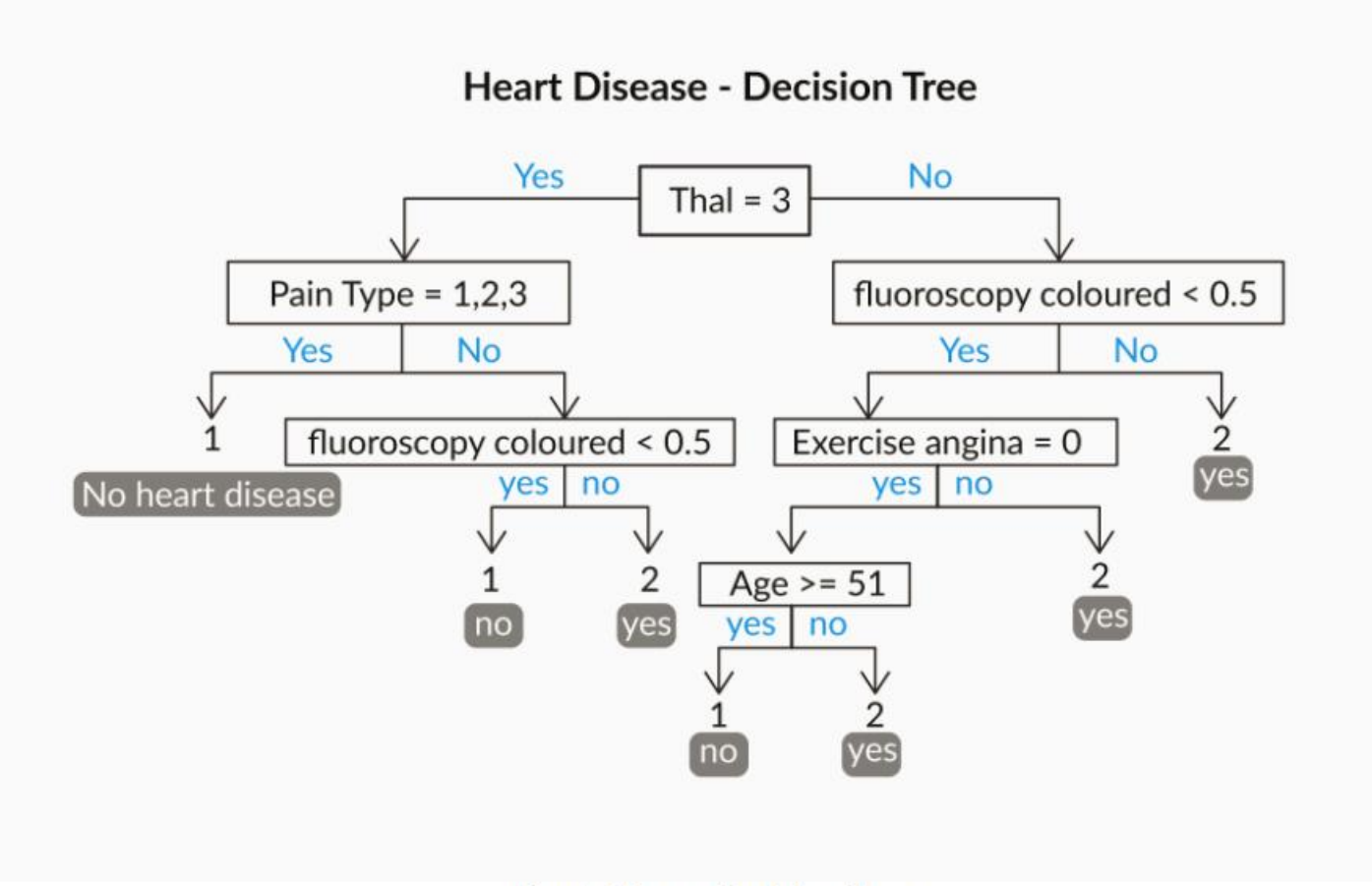
Interpreting a Decision Tree

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. 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.



Heart Disease Decision Tree

Figure: 5

Regression with Decision Trees

There are cases where you cannot directly apply linear regression to solve a regression problem. Linear regression will fit only one model to the entire data set; whereas you may want to divide the data set into multiple subsets and apply linear regression to each set separately.

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.

Homogeneity Measures

In this section we look at the commonly used homogeneity measures used in decision tree algorithms. To illustrate the measures described in this section we use a simple hypothetical example of people in an organization and we want to build a model for who among them plays football. Each employee has two explanatory attributes — Gender and Age. The target attribute is whether they play football. Figure 7 illustrates this dataset — the numbers against P and N indicate the numbers of employees who play football and those who don't respectively, for each combination of gender and age.

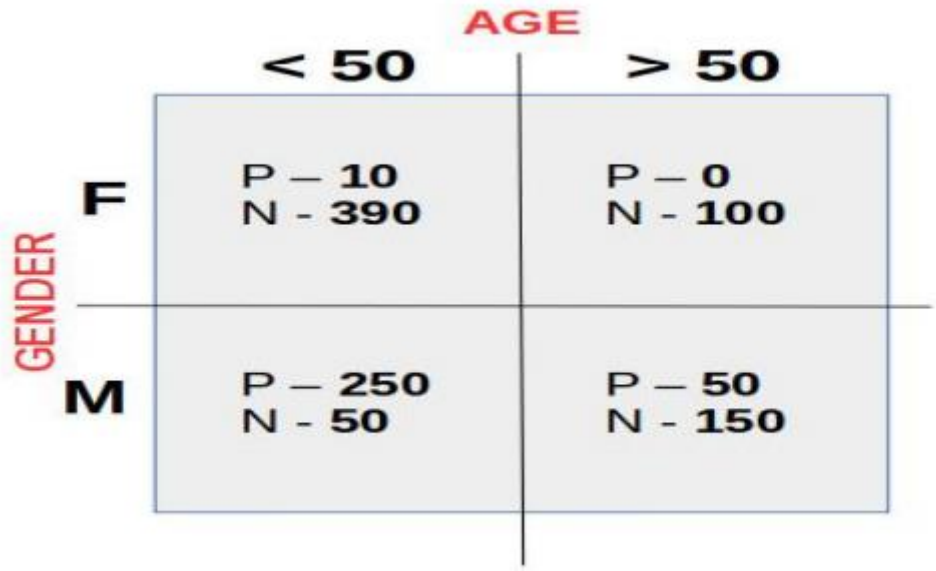


Figure: 6

Gini Index

Gini Index uses the probability of finding a data point with one label as an indicator for homogeneity — if the dataset is completely homogeneous, then the probability of finding a datapoint with one of the labels is 1 and the probability of finding a data point with the other label is zero. An empirical estimate of the probability pi of finding a data point with label i (assuming the target attribute can take say k distinct values) is just the ratio of the number of data points with label i to the total number of data points. It must be that $\int_{i=1}^{k} pi = 1$. For binary classification problems the probabilities for the two classes become p and $(1 - p)$. Gini Index is then defined as:

$$Gini = \sum_{i-1}^{k} Pi^2$$

Note that the Gini index is maximum when Pi = 1 for exactly one of the classes and all others are zero. So higher the Gini index higher the homogeneity. In a Gini based decision tree algorithm, we therefore find the split that maximizes the weighted sum (weighted by the size of the partition) of the Gini indices of the two partitions created by the split. For the example in Figure 6:

- Split on gender: the two partitions will have 10/500 and 300/500 as the probabilities of finding a football player respectively. Each partition is half the total population.

$$Gini = \frac{1}{2}\left(\left(\frac{1}{50}\right)^2 + \left(\frac{49}{50}\right)^2\right) + \frac{1}{2}\left(\left(\frac{3}{5}\right)^2 + \left(\frac{2}{5}\right)^2\right) = 0.7404$$

- Split on Age: the two partitions will have 260/700 and 50/250 as the probabilities, and 700 and 300 as the sizes respectively, giving us a Gini index of:

$$Gini = 0.7\left(\left(\frac{26}{70}\right)^2 + \left(\frac{44}{70}\right)^2\right) + 0.3\left(\left(\frac{1}{5}\right)^2 + \left(\frac{4}{5}\right)^2\right) = 0.5771$$

Therefore we would first need to split on the gender — this split gives a higher GINI index for the partitions. Gini index can only be used on classification problems where the target attribute is categorical.

Information Gain / Entropy-based

The idea is to use the notion of entropy which is a central concept in information theory. Entropy quantifies the degree of disorder in the data. Entropy is always a positive number between zero and 1. Another interpretation of entropy is in terms of information content. A completely homogeneous dataset has no information content in it (there is nothing non-trivial to be learnt from the dataset) whereas a dataset with a lot of disorder has a lot of latent information waiting to be learnt.

Assume the dataset consists of only categorical attributes, both the explanatory variables and the class variable. Again in terms of the probabilities of finding data points belonging to various classes, entropy for a dataset D is defined as

$$\varepsilon[D] = -\sum_{i=1}^{k} Pi \log_2 Pi$$

Notice that the entropy is zero if and only if for some i, pi = 1 and all the other pj = 0 — i.e., when the dataset is completely homogeneous. Consider a k-valued attribute A of the dataset. Suppose we partition the dataset into groups where each group DA=i consists of all the data points for which the attribute A has value i, for each 1 ≤ i ≤ k. The weighted average entropy if we partition the dataset based on the values of A is

$$\varepsilon[D] = \sum_{i=1}^{k}\left(\left(\frac{|\ D_{A=i}\ |}{|D|}\right)\varepsilon\left[D_{A=i}\ \right]\right)$$

This is also the expected entropy of the partition if the dataset is split on the different values of attribute A. This corresponds to a multiway split — partitioning the dataset into groups, each of which is filtered on one value of the splitting attribute. Entropy based algorithms therefore, at each state, find the attribute on which the data needs to be split to make the entropy of the partition minimum.

In practice a slightly modified measure called Information Gain is used. Information Gain, denoted Gain(D, A), is the expected reduction in entropy for the collection of data points D if we filter on a specific value of the attribute A.

Splitting by R-squared

So far, you looked at splits for discrete target variables. But how is splitting done for continuous output variables? You calculate the R2 of data sets (before and after splitting) in a similar manner to what you do for linear regression models. So split the data such that the R2R2 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 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.

Tree Truncation

We have seen earlier that decision trees have a strong tendency to overfit the data. So practical uses of the decision tree must necessarily incorporate some 'regularization' measures to ensure the decision tree built does not become more complex than is necessary and starts to overfit. There are broadly two ways of regularization on decision trees:

- Truncate the decision tree during the training (growing) process preventing it from degenerating into one with one leaf for every data point in the training dataset. One or more stopping criteria are used to decide if the decision tree needs to be grown further.
- Let the tree grow to any complexity. However add a post-processing step in which we prune 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.

We describe some popular stopping criteria and pruning strategies in the following subsections.

Decision Tree Stopping Criteria (Truncation)

There are several ways to truncate decision trees before they start to overfit.

- Minimum Size of the Partition for a Split: Stop partitioning further when the current partition is small enough.
- Minimum Change in Homogeneity Measure: Do not partition further when even the best split causes an insignificant change in the purity measure (difference between the current purity and the purity of the partitions created by the split).
- Limit on Tree Depth: If the current node is farther away from the root than a threshold, then stop partitioning further.
- Minimum Size of the Partition at a Leaf: If any of partitions from a split has fewer than this threshold minimum, then do not consider the split. Notice the subtle difference between this condition and the minimum size required for a split.
- Maxmimum number of leaves in the Tree: If the current number of the bottom-most nodes in the tree exceeds this limit then stop partitioning.

Decision Tree (Post)-Pruning

One popular approach to pruning is to use a validation set — a set of labelled data points, typically kept aside from the original training dataset. This method called reduced-error pruning, considers every one of the test (non-leaf) nodes for pruning. Pruning a node means removing the entire subtree below the node, making it a leaf, and assigning the majority class (or the average of the values in case it is regression) among the training data points that pass through that node. A node in the tree is pruned only if the decision tree obtained after the pruning has an accuracy that is no worse on the validation dataset than the tree prior to pruning. This ensures that parts of the tree that were added due to accidental irregularities in the data are removed, as these irregularities are not likely to repeat.

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 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.
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.

Building Decision Trees in Python

```
# Importing decision tree classifier from sklearn library
from sklearn.tree import DecisionTreeClassifier

# Fitting the decision tree with default hyperparameters
dt_default = DecisionTreeClassifier()
dt_default.fit(X_train, y_train)
```

```
# Let's check the evaluation metrics of our default model
# Importing classification report and confusion matrix from sklearn metrics
from sklearn.metrics import classification_report, confusion_matrix

# Making predictions
y_pred_default = dt_default.predict(X_test)

# Printing classification report
```

```
print(classification_report(y_test, y_pred_default))

# Printing confusion matrix
print(confusion_matrix(y_test,y_pred_default))
```

Building Decision Trees in Python - Hyperparameter Tuning

Tuning max_depth

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


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

# parameters to build the model on
parameters = {'max_depth': range(1, 40)}

# instantiate the model
dtree = DecisionTreeClassifier(criterion = "gini", random_state =100)

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

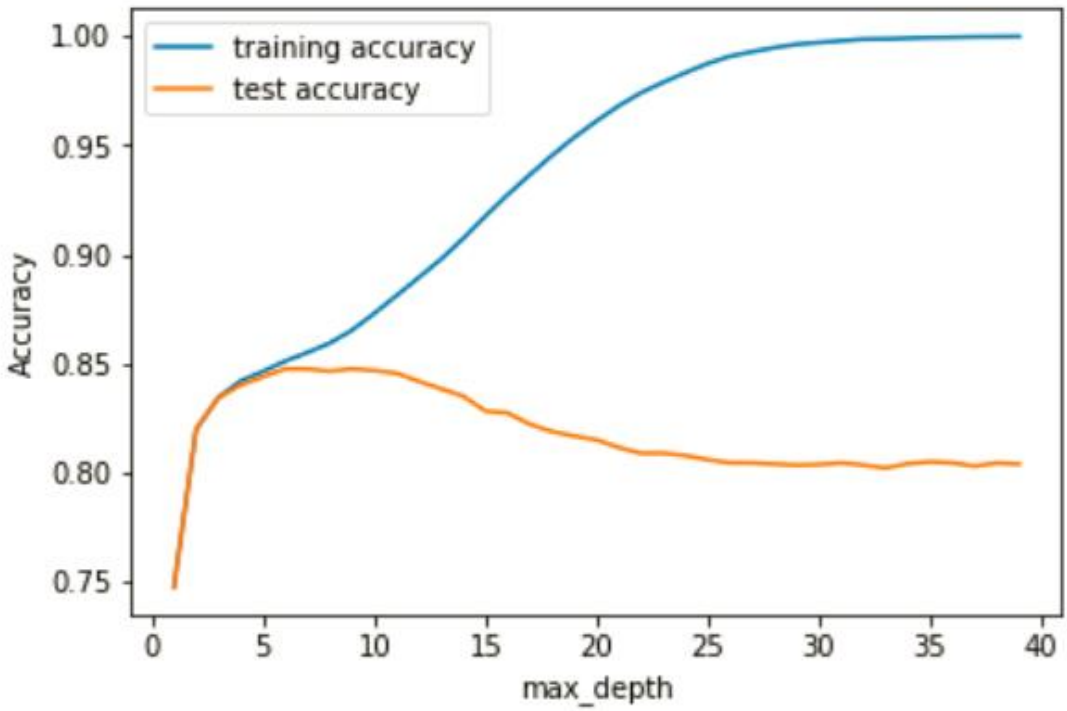


Figure: 7

You can see that as we increase the value of max_depth, both training and test score increase till about max-depth = 10, after which the test score gradually reduces. Note that the scores are average accuracies across the 5-folds.

Thus, it is clear that the model is overfitting the training data if the max_depth is too high. Next, let's see how the model behaves with other hyperparameters.

Tuning min_samples_leaf

The hyperparameter **min_samples_leaf** indicates the minimum number of samples required to be at a leaf.

So if the values of min_samples_leaf is less, say 5, then the will be constructed even if a leaf has 5, 6 etc. observations (and is likely to overfit).

Let's see what will be the optimum value for min_samples_leaf.

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


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

```
# parameters to build the model on
parameters = {'min_samples_leaf': range(5, 200, 20)}

# instantiate the model
dtree = DecisionTreeClassifier(criterion = "gini", random_state =100)

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

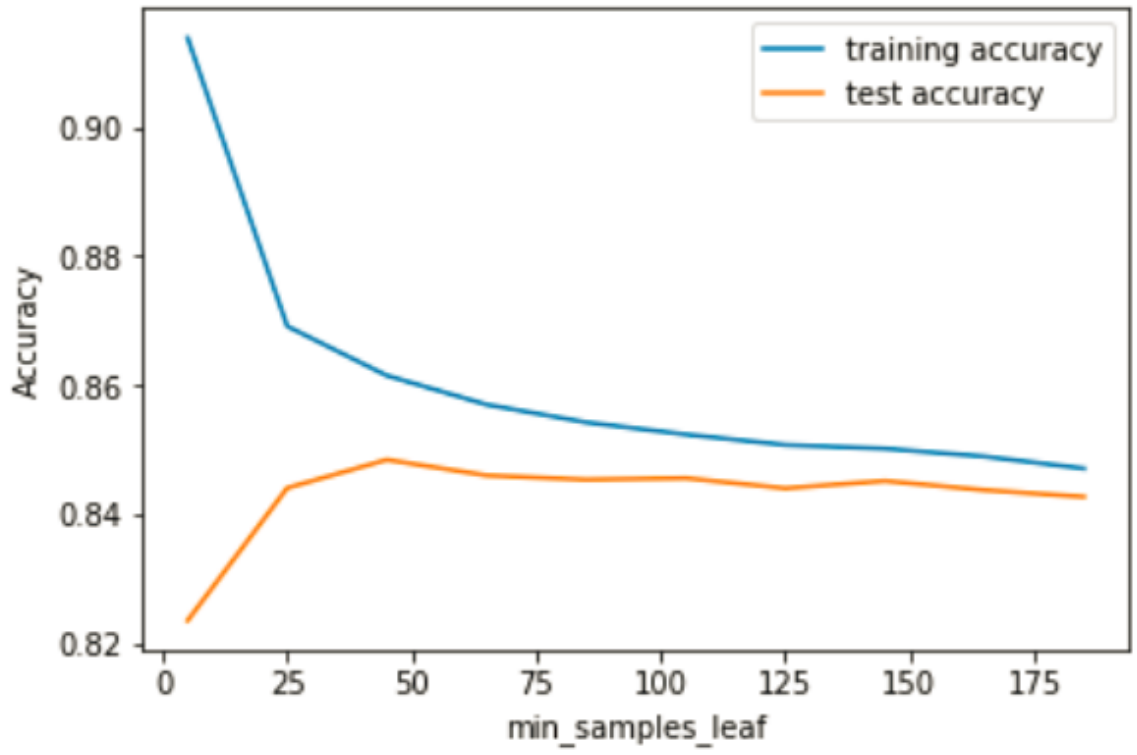


Figure: 8

Tuning min_samples_split

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 furthur divided into leaf nodes.

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


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

```
# parameters to build the model on
parameters = {'min_samples_split': range(5, 200, 20)}

# instantiate the model
dtree = DecisionTreeClassifier(criterion = "gini", random_state =100)

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

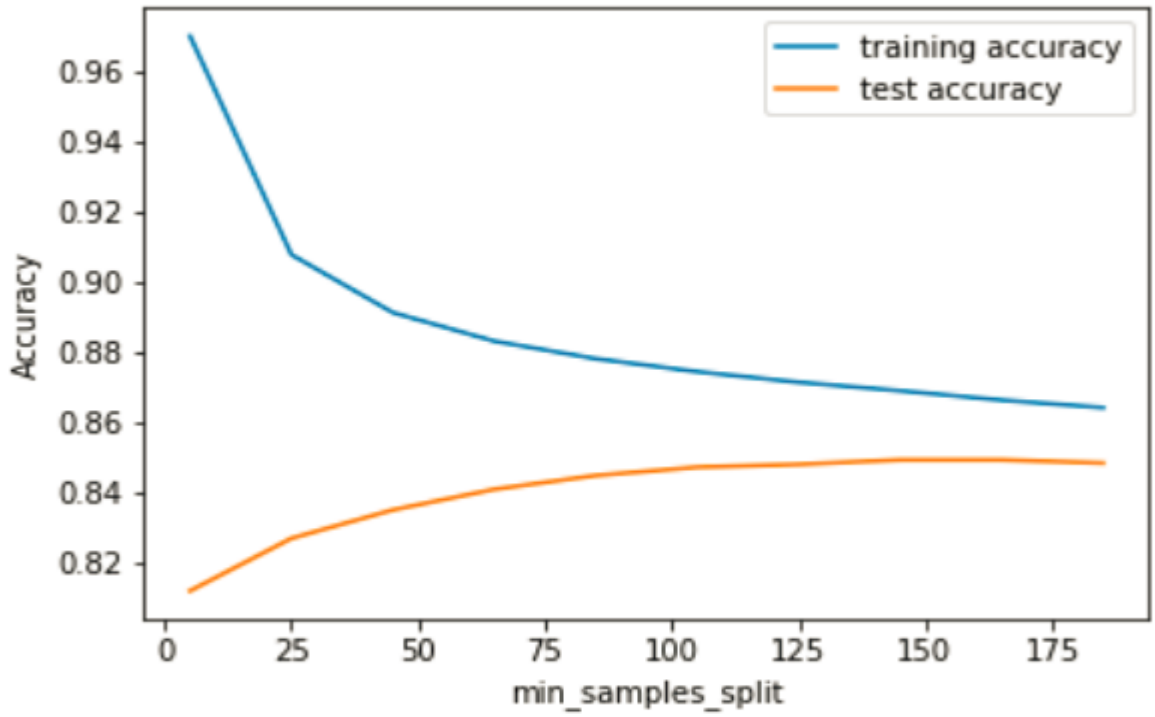


Figure: 9

Grid Search to Find Optimal Hyperparameters

We can now use GridSearchCV to find multiple optimal hyperparameters together. Note that this time, we'll also specify the criterion (gini/entropy or IG).

```
# Create the parameter grid
param_grid = {
    'max_depth': range(5, 15, 5),
    'min_samples_leaf': range(50, 150, 50),
    'min_samples_split': range(50, 150, 50),
    'criterion': ["entropy", "gini"]
}
```

```
n_folds = 5

# Instantiate the grid search model
dtree = DecisionTreeClassifier()
grid_search = GridSearchCV(estimator = dtree, param_grid =param_grid,
                           cv = n_folds, verbose = 1)

# Fit the grid search to the data
grid_search.fit(X_train,y_train)
```

Running the model with best parameters obtained from grid search.

```
# model with optimal hyperparameters
clf_gini = DecisionTreeClassifier(criterion = "gini",
                                  random_state = 100,
                                  max_depth=10,
                                  min_samples_leaf=50,
                                  min_samples_split=50)
clf_gini.fit(X_train, y_train)

# accuracy score
clf_gini.score(X_test,y_test)

# plotting the tree
dot_data = StringIO()
export_graphviz(clf_gini,
out_file=dot_data,feature_names=features,filled=True,rounded=True)

graph = pydot.graph_from_dot_data(dot_data.getvalue())
Image(graph[0].create_png())
```