Decision Trees

Table of Contents

[Purity of Splits 3](#_Toc106495361)

[Test Functions 4](#_Toc106495362)

[Univariate Trees 5](#_Toc106495363)

[Impurity Measure 6](#_Toc106495364)

[Split and Sub-Tree Generation 8](#_Toc106495365)

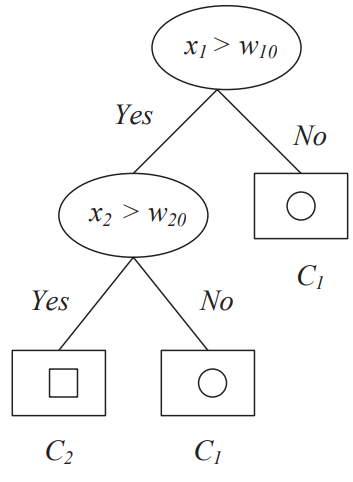
[Important Factors 10](#_Toc106495366)

[Classification Tree 11](#_Toc106495367)

[Regression Tree 15](#_Toc106495368)

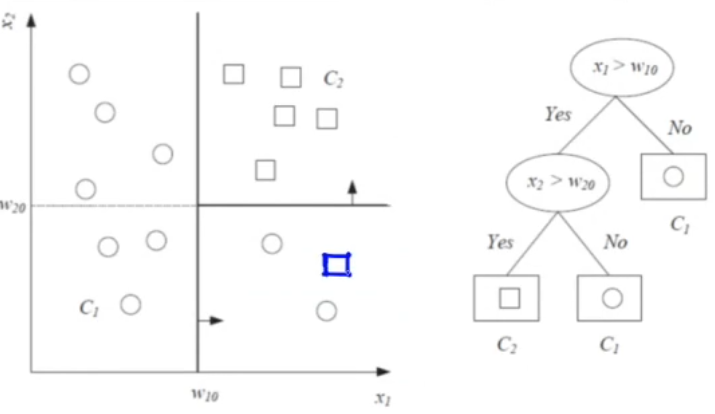
[Error Thresholds 17](#_Toc106495369)

A **decision tree** is a hierarchical data structure that uses the **divide and conquer** strategy. It is efficient and has **no parameters**, plus it can be used to solve both classification and regression problems.



A decision tree, such as the one shown above, consists of **decision nodes** and **terminal leaves**. The goal is to create a model that predicts values based on simple decision rules. The advantage is that the tree can be converted into a set of if-then statements, as in if a decision node is taken, then the sub-branches can be explored. This characteristic is called **interpretability**.

The process of constructing a decision tree from training data is called **tree induction**. There are several ways to correctly code a decision tree from a single set of training data.



Consider the training data and the corresponding decision tree above. For a given input, at each decision node, one of the branches is taken based on the outcome of the decision. At each decision node, , a **test function**, , is implemented which has discrete outcomes for each of the branches. This process starts at the root and is repeated recursively until a **leaf node** is hit, at which point the outcome is found.

## Purity of Splits

In this tree, we started by checking the value of and then checked the value of . However, we could also have checked first and then . So why did we not do that?

Every time we break the tree into branches from a decision node, we are said to be creating a **split**. The way we setup our tree depends on how **pure** the splits are. A single sub-branch from a split is said to be completely pure if there is only one possible outcome from that sub-branch. We do not divide that sub-branch any further, i.e. we have reached a leaf node. Our goal is to maximize the purity of each split.

For the above case, if we had checked first and then , then we would get multiple sub-branches that are not pure, instead of just getting 1 sub-branch that is not pure. There are also the details of choosing a threshold value, since we need to pick the appropriate threshold value as well to maximize the purity of our splits. We will be looking into the details of how to calculate the purity of a split and thus take decisions about how to setup the tree later.

Even for the case where we check the value of first and then , the second split from the decision node results in two sub-branches, both of which are impure. However, that is still the best-case scenario for us. This means that we are not guaranteed to always get a completely pure sub-branch from a split. It is possible to have all our sub-branches be impure, in which case we can choose to keep splitting them further.

## Test Functions

The test function, , defines a discriminant in the -dimensional input space, dividing it into smaller regions. We can create different models for , which will give us different shapes and decision boundaries.

The hierarchical placement of the decisions allows us to have **fast localization**, i.e. we can reach a conclusion quickly. In the best case, the correct result can be found in decisions.

## Univariate Trees

In a **univariate tree**, each decision node tests just one of the input dimensions (attributes), . At each node, for numerical data, we are able to define a **threshold** value that allows us to have a binary split. However, if our data is **discrete**, e.g. is red, green, blue, or yellow, the split will have sub-branches for the possible options.

At each decision node, we have to firstly pick the input dimension that gives us the best possible split, and for numerical data, we also have to pick the threshold value that gives us the best possible split. Since we are trying to find the best possible split at each decision node, tree learning algorithms are **greedy algorithms**.

Example



For the above data, one possible tree is the one given below.



As can be seem here, we will sometimes end up with trees that do not even use some of the features. In this example, the Temperature feature was not use. In these cases, we can **discard** those features.

## Impurity Measure

For decision trees that are used for classification, called **classification trees**, the goodness of a split is quantified using an **impurity measure**.

For 6 samples, having a branch with 3 positive and 3 negative samples in it is the worst case scenario. In any other case, we will have a lower impurity measure. ON the flip side, the best-case scenario would be to have all 6 samples in either the positive or the negative class. This can be understood to essentially mean the following graph:



This graph is for the **entropy function** specifically for a two-class problem.

For a particular decision node , if we have samples in total and the number of samples in the th class is , then the total impurity for the split is given by

where

For a binary split, we can also use

where is for the positive class.

This gives us a set of properties:

* for any given , since the highest entropy occurs at .
* , since placing all the samples in the same class gives us a entropy.
* is increasing as we move from to and decreasing as we move from to .

There are two other forms of the impurity test:

1. The **Gini Index**
2. The **Misclassification Error**

## Split and Sub-Tree Generation

We will now discuss how to actually split the tree. If a particular node is impure, we should split it to decrease the impurity. There can be multiple different attributes to choose from, and if the attribute is numeric, there are multiple threshold values at which we can split it.

Among all the possible choices, we should take the split that **minimizes the impurity** after the split. This process is locally optimal, and it does not guarantee that we will get the smallest decision tree. The process of finding the smallest tree is an NP-Complete problem (which basically means we cannot be 100% certain that we have found it). The size of the tree is measured in terms of the number of nodes as well as the complexity of the decisions at each node.

At a particular node , if the attribute we are looking at has the class (e.g for the attribute colour, could be red, blue, green, etc.), then the probability that the outcome is of class is given by

and the total impurity after the split is given by

Thus, we are getting to a node and trying to decide which attribute to use to split it. For each attribute, we are computing the total impurity for all possible sub-branches ( values) using each data point in the sub-branches ( values). We will choose the attribute which gives us the minimum possible impurity.

If we have data points, there are possible positions at which we can split the data points. This can be a huge number of splits to test out. Because of this, it is best to test splits between adjacent points that belong to **different classes**.

For two separate sub-branches that are impure, it is even possible to carry out the further splitting **parallelly**, since the two sub-branches are independent to each other.

Some common splitting algorithms are:

* CART
* ID3
* C4.5

## Important Factors

One issue with the splitting process is that it favours attributes that have **many classes**. Consider that we have an attribute with 10 possible classes and we end up getting sub-branches with 1 record in each sub-branch. This will give us 0% impurity. However, it is also making the tree more **complex**. To avoid this, we can provide a condition that each sub-branch must have a minimum number of records in it for the attribute to be considered.

Another issue is that **noise** will cause the tree to become very large and also cause us to start overfitting. For example, if we have a sub section with 10 data points, 9 of which are from one class and the last data point is from a different class, it makes sense for us to be able to ignore that last data point. It is noise. However, under the splitting method, we will also try to split this (which will definitely decrease impurity), but in the process we will end up overfitting to the training data and creating an unnecessarily large tree. To deal with this issue, we can set a threshold value beyond which we will not split. For example, if the purity value is at least 80%, we can use the major class and stop splitting. However if we choose to do this, it is best to store the purity value since we are not 100% certain of the class we are assigning.

## Classification Tree

Some pseudocode that can be used to generate a classification tree is provided below:

GenerateTree(X)  
 If NodeEntropy(X) < /\* equation 9.3 \*/  
 Create leaf labelled by majority class in X  
 Return  
 i ← SplitAttribute(X)  
 For each branch of xi  
 Find Xi falling in branch  
 GenerateTree(Xi)  
  
SplitAttribute(X)  
 MinEnt ← MAX  
 For all attributes i = 1, ..., d  
 if xi is discrete with n values  
 Split X into Xi, ..., Xn by xi  
 e ← SplitEntropy(X1, ..., Xn) /\* equation 9.8 \*/  
 if e < MinEnt MinEnt ← e; bestf ← i  
 else /\* xi is numeric \*/  
 For all possible splits  
 Split X into X1, X2 on xi  
 e ← SplitEntropy(X1, X2)  
 if e < MinEnt MinEnt ← e; bestf ← i  
 Return bestf

PSEUDOCODE

Example

Consider that we have the following data:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Outlook** | **Temperature** | **Humidity** | **Windy** | **PlayGolf** |
| Rainy | Hot | High | FALSE | No |
| Rainy | Hot | High | TRUE | No |
| Overcast | Hot | High | FALSE | Yes |
| Sunny | Mild | High | FALSE | Yes |
| Sunny | Cool | Normal | FALSE | Yes |
| Sunny | Cool | Normal | TRUE | No |
| Overcast | Cool | Normal | TRUE | Yes |
| Rainy | Mild | High | FALSE | No |
| Rainy | Cool | Normal | FALSE | Yes |
| Sunny | Mild | Normal | FALSE | Yes |
| Rainy | Mild | Normal | TRUE | Yes |
| Overcast | Mild | High | TRUE | Yes |
| Overcast | Hot | Normal | FALSE | Yes |
| Sunny | Mild | High | TRUE | No |

At the root node, we know that we played golf 9 out of 14 days, so the entropy values is . Next, we need to pick an attribute to split by.

For outlook:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | | PlayGolf (14) | |  |
| Yes | No |  |
| Outlook | Sunny | 3 | 2 | 5 |
| Overcast | 4 | 0 | 4 |
| Rainy | 2 | 3 | 5 |

Similarly, , and .

Thus, it becomes obvious that we should split by the Outlook.

Another way of looking at this is the **Information Gain**. This is the difference between the entropy before and after the split.

Once we have performed the first split, it is time to move on to the second one.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Outlook** | **Temperature** | **Humidity** | **Windy** | **PlayGolf** |
| Sunny | Mild | Normal | FALSE | Yes |
| Mild | High | FALSE | Yes |
| Cool | Normal | FALSE | Yes |
| Cool | Normal | TRUE | No |
| Mild | High | TRUE | No |
| Overcast | Hot | High | FALSE | Yes |
| Mild | High | TRUE | Yes |
| Hot | Normal | FALSE | Yes |
| Cool | Normal | TRUE | Yes |
| Rainy | Hot | High | FALSE | No |
| Hot | High | TRUE | No |
| Mild | High | FALSE | No |
| Cool | Normal | FALSE | Yes |
| Mild | Normal | TRUE | Yes |

For each possible branch of the Outlook attribute, we need to calculate the different entropy values for all the other attributes.

## Regression Tree

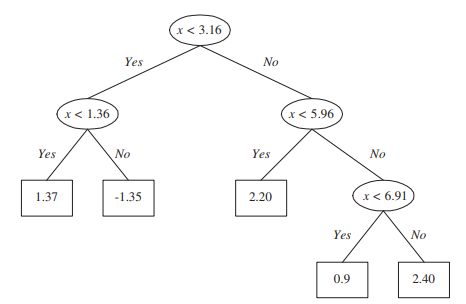
The processes of generating a regression tree is exactly the same as that for a classification tree. The only difference is that the **entropy calculation** is replaced with a new formula more appropriate to regression problems.

where

and .

Here, is the sample number, is the actual value of the data point and is the estimated value (from the line of best fit).

Examining a regression tree should make this easier to picture.



Basically, instead of having a single line, we are using multiple lines based on the value of .

At node , we have records. Let be the number of records that take the th branch. Thus,

and

The error after the split is given by

### Error Thresholds

As previously discussed, we also need to have a threshold beyond which we will not split the tree further. For different threshold values, we will get different accuracy levels.

