Nikola Pulev

Machine Learning with Decision Trees and Random Forests

Course Notes

365√DataScience

Table of Contents

Abstract

A Decision tree is a supervised classification and regression machine learning algorithm. It is famous for being one of the most intuitive and easy to understand methods. This makes it, as well, a good starter algorithm to learn the quirks of the specific dataset and problem one is trying to solve. When it comes to actual results, though, decision trees have another trick up their sleeve – they can be stacked together to form a random forest, that can outperform many other methods.

The following notes serve as a complement to the "Machine Learning with Decision Trees and Random Forests" course. They list the algorithms' pros and cons, outline the working of the decision trees and random forests algorithms, cover in greater detail the more involved topic of Gini impurity and entropy, and summarize the most commonly used performance metrics.

Keywords: machine learning algorithm, decision tree, random forest, classification, gini impurity, information gain, pruning, ensemble learning, bootstrapping, confusion matrix, accuracy, precision, recall, F_1 score

1 Motivation

In this section, we summarize the advantages and disadvantages of both algorithms.

1.1 Decision Trees

To the naked eye, decision trees might look simple at first glance – in fact, they may look way *too* simple to be remotely useful. But it is that simplicity that makes them useful. Since in today's world it is extremely easy to create very complex models with just a few clicks, many data scientist cannot understand nor explain what their model is doing. Decision trees, while performing averagely in their basic form, are easy to understand and when stacked reach excellent results.

Table 1: Pros and cons of Decision trees.

Pros	**Cons**
Intuitive	Average results
Easy to visualize and interpret	Trees can be unstable with respect to the train data
In-built feature selection	Greedy learning algorithms
No preprocessing required	Susceptible to overfitting (there are measures to counter this)
Performs well with large datasets	
Moderately fast to train and extremely fast to predict with	

1.2 Random Forests

Random forests are built upon many different decision trees, however, with different measures set in place to restrict overfitting. Thus, they obtain higher performance.

Table 2: Pros and cons of Random Forests.

Pros	**Cons**
Gives great results	Black box model - looses the interpretability of a single decision tree
Requires no preprocessing of the data	Depending on the number of trees in the forest, can take a while to train
Automatically handles overfitting in most cases	Outperformed by gradient-boosted trees
Lots of hyperparameters to control	
Performs well with large datasets	

2 Computer Science Background: Trees

In order to discuss decision trees, we have to clear up the meaning of 'tree' in a programming context. In computer science, a tree is a specific structure used to represent data. It might look something like this.

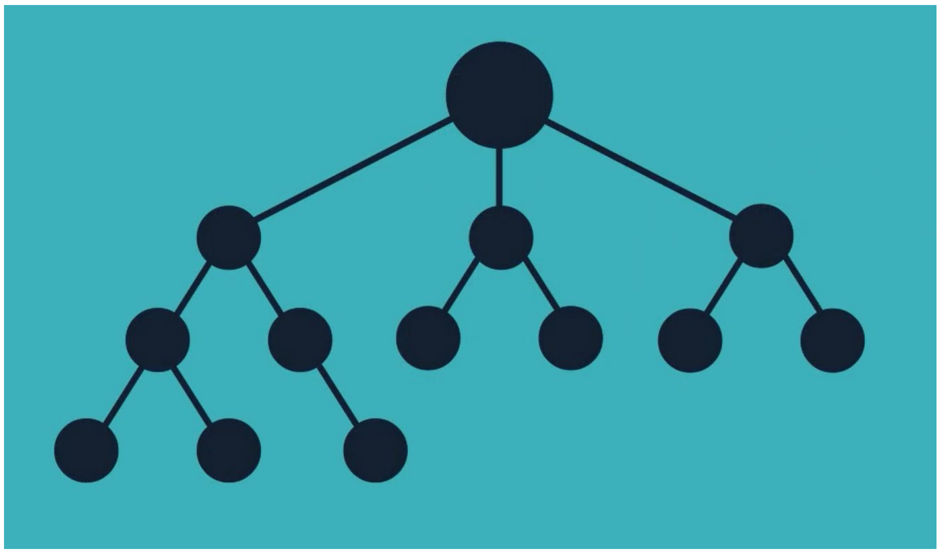


Figure 1: A typical tree in computer science

From this picture, it's clear how the name came about – it definitely does look like an upside-down tree, branching more and more as you go down. Now, there are 2 main elements that make up the tree – nodes and edges/branches.

- **Nodes** are the black circles in the picture above. They contain the actual data. This data is, generally, not restricted to a particular type.
- **Edges** are the black lines connecting the different nodes. They are often called **branches**.

You might recognize these two elements from a different mathematical structure – graph. And that's entirely correct, you can think of the tree as a graph with additional restrictions.

Those restrictions are that a node can be connected to a different node that is either one level higher, or one level lower. Moreover, every node, except the very first one, should be connected to exactly one node higher up. These rules mean that connections such as the ones illustrated below are not permitted.

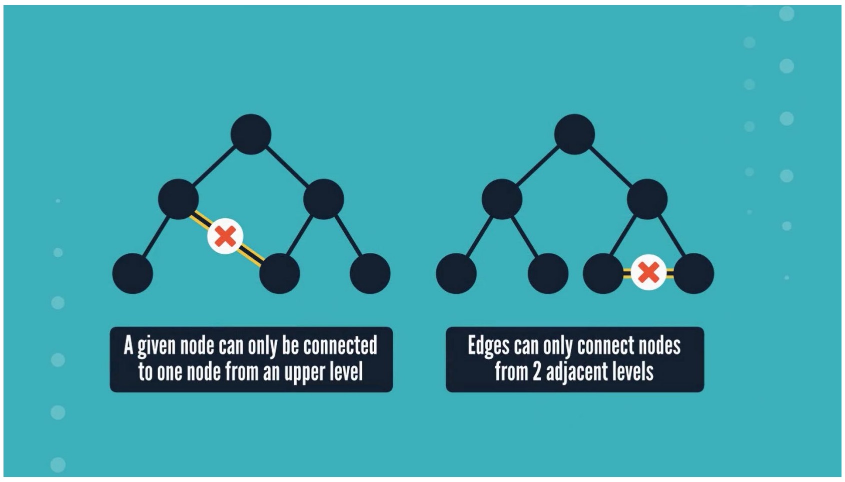

A given node can only be connected to one node from an upper level
Edges can only connect nodes from 2 adjacent levels

Figure 2: The highlighted connections are forbidden in a typical tree structure

From the pictures so far, we can see that a tree is a structure with the pattern of a node, connected to other nodes through edges, repeated again and again recursively to create the whole tree. Thus, it is a good idea to be able to distinguish between the different parts. First, it's crucial to remember that a tree has a well-defined hierarchy and we always view it from top to bottom. Then, we can identify the following elements:

- **Root node** – this is the uppermost node, the one that starts the tree
- **Parent node** – when considering a subset of the tree, the parent node is the one that is one level higher and connects to that subtree (see *figure 3*)
- **Child node** – when given a node, the ones stemming from it, one level lower, are its children (see *figure 3*)

- **Leaf node** – A node that has no children. This is where the tree terminates (Note that a tree can terminate at different points on different sides)
- **Height** – how many levels the tree has. For example, we can say that the tree from *figure 1* has a height of 4
- **Branching factor** – this signifies how many children there are per node. If different nodes have different number of children, we can say that the tree has no definitive branching factor. In principle, there can exist trees with however big branching factor you want. However, an extremely popular tree subvariant has 2 branches per node at most. This type of tree is called a **binary tree**.

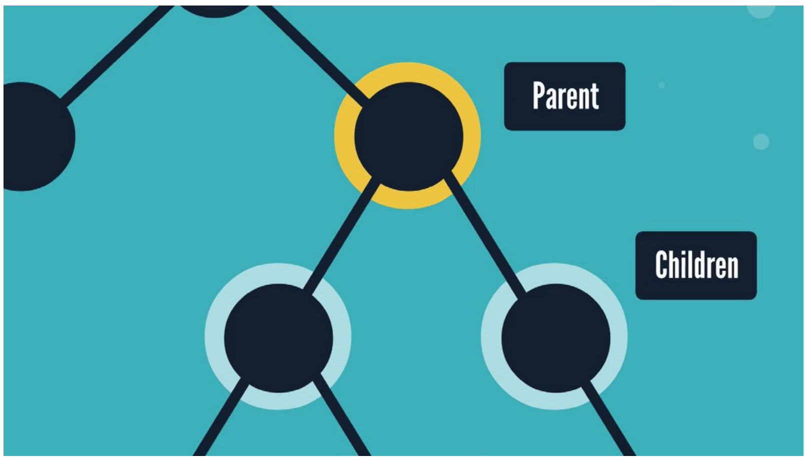


Figure 3: The relationship between a parent and children nodes

One very common use of the binary tree is the binary search trees which are used for efficient implementations of searching and sorting algorithms. There are, of course, many others, including decision trees.

3 What is a decision tree?

Decision trees are a common occurrence in many fields, not just machine learning. In fact, we commonly use this data structure in operations research and decision analysis to help identify the strategy that is most likely to reach a goal. The idea is that there are different questions a person might ask about a particular problem, with branching answers that lead to other questions and respective answers, until they can reach a final decision. But, this is a very visual topic, so let's just see 2 examples of that:

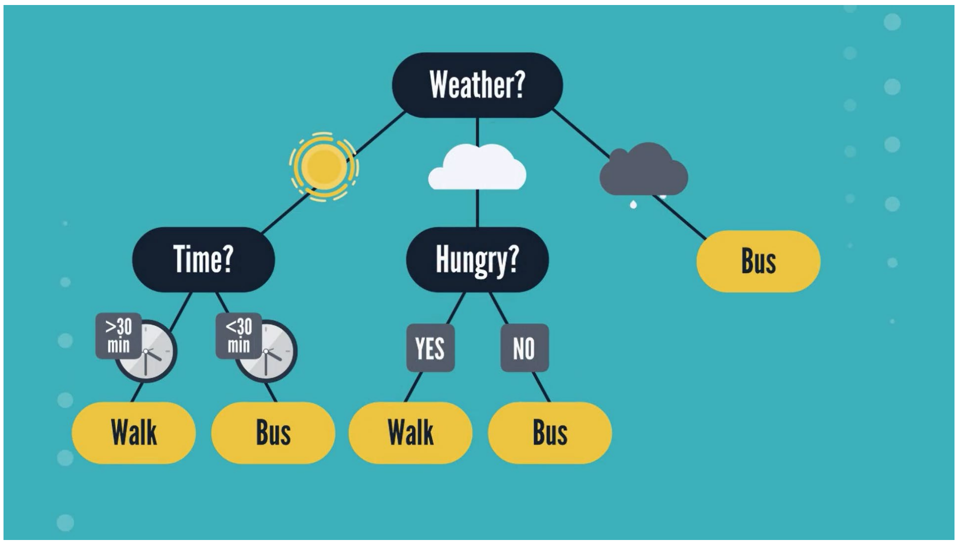


Figure 4: Decision tree example about method of transportation based on weather

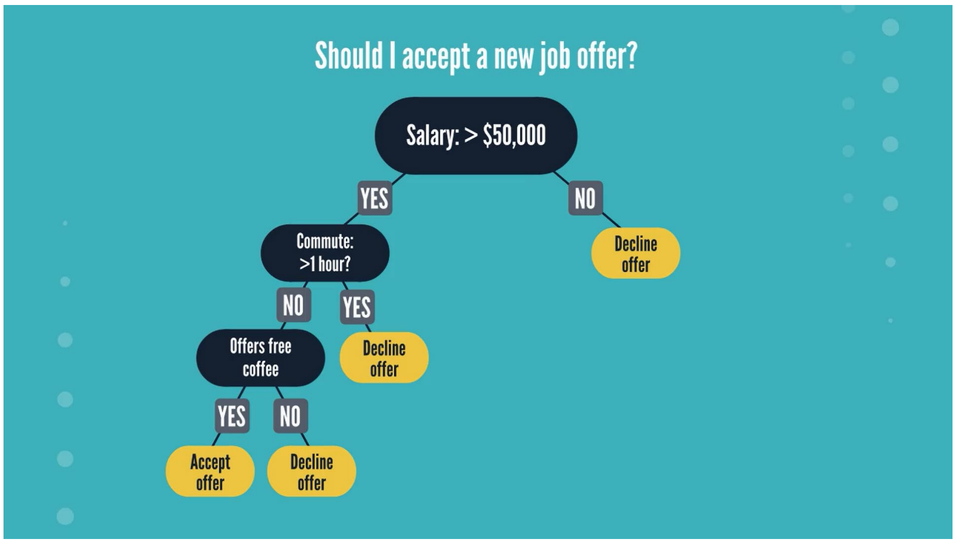


Figure 5: Decision tree example based on whether to accept a job offer

As can be seen from those examples, the nodes in a decision tree hold important questions regarding the decision one wants to make. Then, the edges/branches represent the possible answers to those questions. By answering the different questions and following the structure down, one arrives at a leaf node (marked yellow in the above illustrations) that represents the outcome (decision).

The decision trees so far, however, do not represent a machine learning problem. How would an ML decision tree look like? Well, here it is:

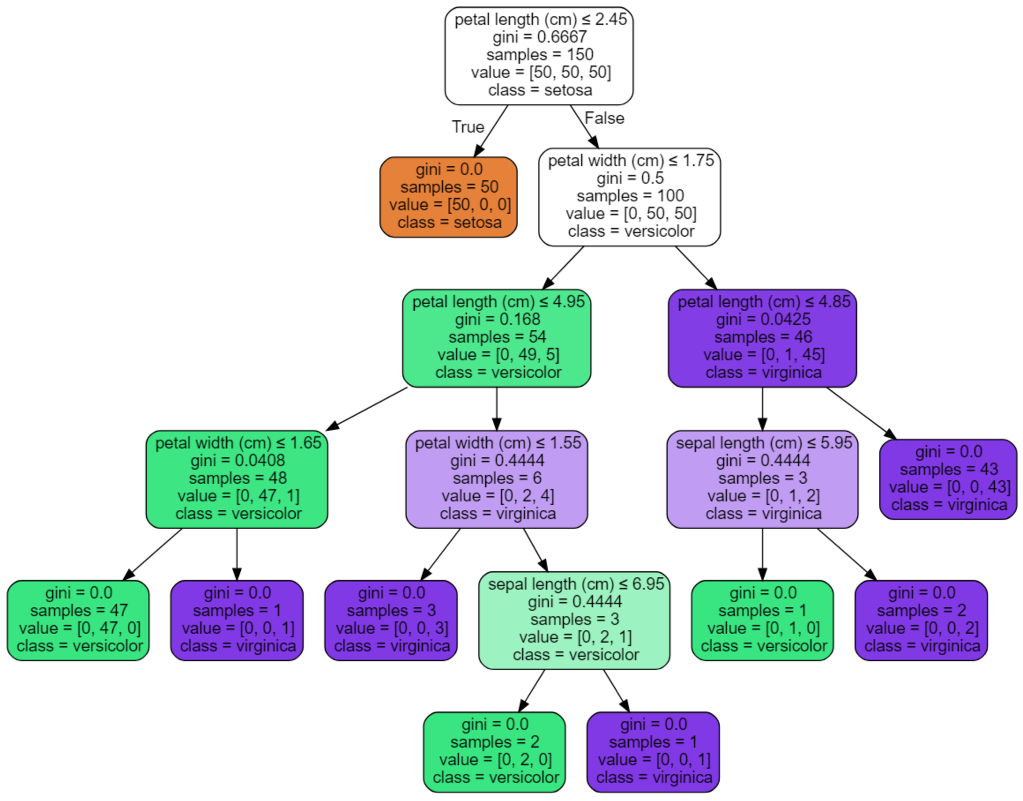


Figure 6: ML decision tree based on the Iris dataset. The dataset features flower petal and sepal dimensions with the objective to predict the exact flower species

This is a real tree trained on the Iris dataset. The input features are the sizes of the petals and sepals of different Iris flowers, with the objective to classify the them into 3 Iris species. The nodes now carry a lot more information, most of which is just informative to the reader, not part of the "questions". In machine learning context, the decision tree asks questions regarding the input features themselves (most often the question is whether the value is bigger or smaller than some threshold).

And here lies the usefulness of decision trees. As they can be easily visualized, they offer data scientists tools to analyze how model makes predictions. Moreover, since the tree is a hierarchical structure, the higher a certain node is, the more important it is to the problem at hand. Thus, we can say that decision trees incorporate feature selection automatically.

From the above tree, we can extract another important term for decision trees – **split**. Let's consider what happens to our training set when we apply it to a decision tree? Since it consists of many different data points with different feature values, we would expect that some of the points follow the left arrow, while others follow the right one. Thus, we have effectively split our dataset in two. Then, each of the parts is further chopped in two at every subsequent node. That's why, a node is often referred to as a **split** during training.

So, what types of decision trees are there?

Well, decision trees can solve both **regression** and **classification** problems. Popular implementations of the algorithm include ID3, C4.5 and CART.

CART (**C**lassification **A**nd **R**egression **T**ree) is especially important here as it is the algorithm that sklearn chose in order to implement decision trees.

4 How are decision trees constructed?

While the decision tree itself is easy to understand, the process that creates it is slightly more complicated. Nevertheless, there are a couple of main points that we can discuss. So, let's take a look at them.

Decision trees are generated through **greedy algorithms**. Greedy algorithms are ones that work by choosing the best option available right now, without considering the whole picture. Therefore, they are fast, as there's no need to go through all of the possibilities, but can sometimes produce suboptimal solutions. Take, as an example, the problem of finding the shortest path between two cities. If you take the shortest street at every junction, you may end up in a situation where you are actually going away from the desired destination. So, a true optimal solution can be reached only if you consider all the roads between you and the destination, not just the ones at each junction.

Nevertheless, the greedy algorithms do a good enough job at finding a solution as close as possible to the optimal one, that this doesn't matter. What matters is that they are fast.

So, these algorithms construct the tree one node at a time. During the process, they look at the tree so far, and decide which node would best separate the data. In other words, they look for the best way to **split** the data at each node.

Here, the word "best" is subjective, so we need to assign concrete meaning behind it. In decision trees, this is done by defining different metrics that quantify how "good" or "bad" a certain split is. The algorithm simply tries to minimize or maximize those. So, the real important part are the actual metrics. The two most popular ones are gini impurity and information gain (entropy). Let’s take a look at those.

4.1 Gini impurity

Before we dive in, it is worth noting what really matters for the algorithms – the number of samples from each class that is present in the node. They don’t look at the entire tree, but rather take it one node at a time. In this context, a node may be called a “split”, since we split our data in 2 – for Yes or No. Of course, some splits are more useful than others. For instance, one that funnels all of the data to the left branch and none of it to the right might be a really bad split, since we haven’t actually changed anything. The metrics’ job is to quantify exactly how good or bad a certain split is. Gini impurity is one such metric.

The formula for Gini impurity is:

$$Gini = \sum_{i}^{n} p_i(1-p_i) = 1 - \sum_{i}^{n} (p_i)^2,$$

where p_i is the proportion of samples in the i-th class with respect to the whole set of data. The data considered here, is the data present at the node we are computing this metric for. This will be the whole dataset for the root node, but it will necessarily become a smaller and smaller subset the deeper down the tree we go.

Gini impurity (named after Italian mathematician Corrado Gini) is a measure of how often a randomly chosen element from the set would be incorrectly labeled if it was randomly labeled according to the distribution of labels in the subset. In simple words, the idea is to find how much would change in the node if we randomly shuffle our data. Allow me to illustrate with an example.

Let’s say we have a dataset with 2 classes – red and blue. Imagine we have a node that contains 10 data points. Now, suppose these samples have the following class distribution – 3 of them are red, and 7 are blue.

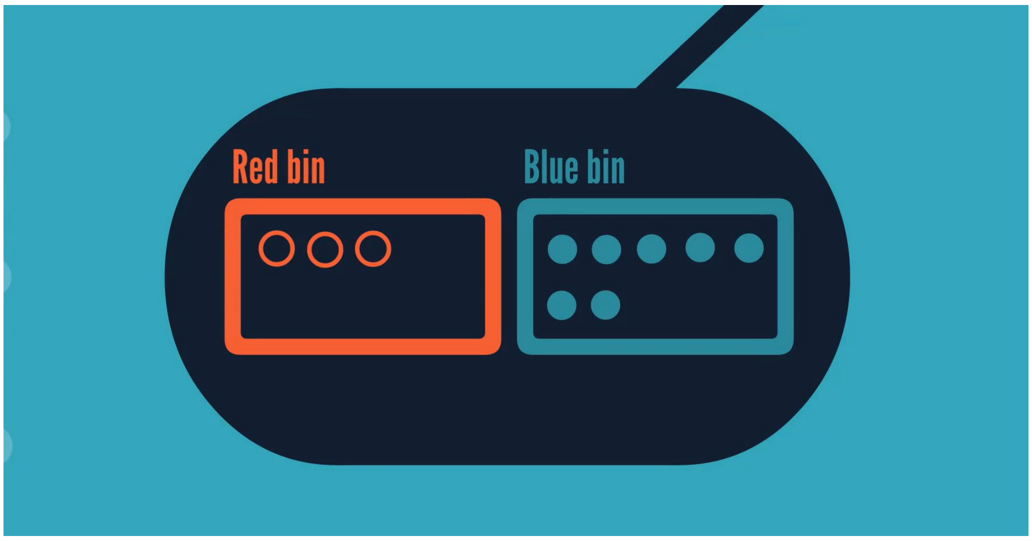


Figure 7: Data inside a node - 3 data points of the red class and 7 data points of the blue class

So far, every data point is in the correct bin, so we might say that we have no misclassifications.

In this case, gini impurity will try to measure how the accuracy will change if we randomly shuffled those 10 samples. We still need to have 3 samples in the red bin and 7 samples in the blue bin. However, the bins are no longer guaranteed to contain only red and blue data points, respectively. In other words, there is misclassification of some of the data. Essentially, we use this measure to identify what the misclassification rate is. The bigger it is, the bigger the gini impurity. The algorithms, thus, try to **minimize** the gini impurity. The smallest possible gini is 0 and it is achieved when all samples in the node are of a single class.

So, a random shuffle the data above may look like this:

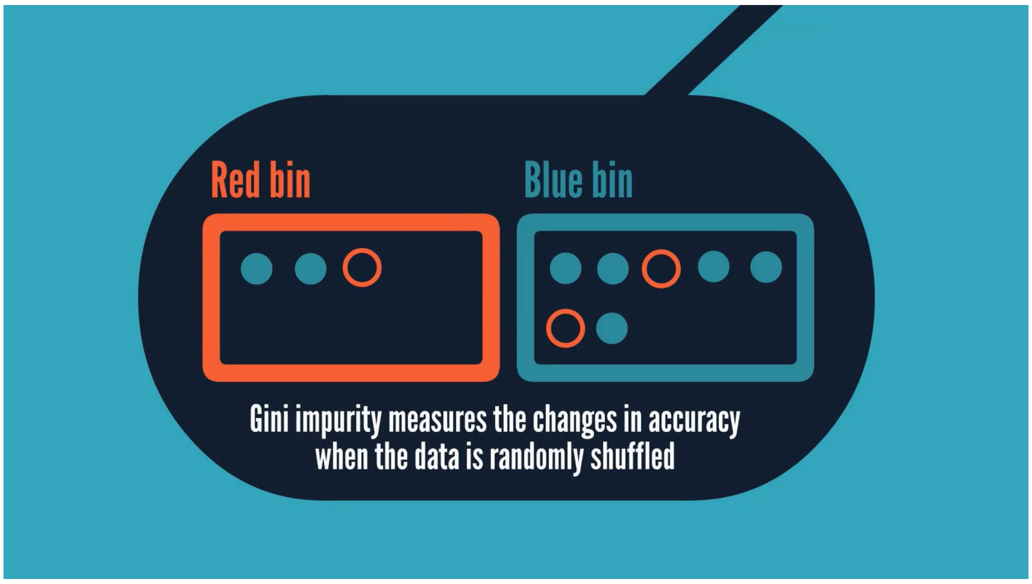

Red bin
Blue bin
Gini impurity measures the changes in accuracy when the data is randomly shuffled

Figure 8: The same data in the node, but now randomly shuffled. Notice that there is some "misclassification".

Now, we can see that not all red points are in the red bin. Likewise for the blue data points. Thus, there is some misclassification. Gini impurity tries to measure the average rate of precisely this misclassification.

So, how does this relate to the formula expressed above. Well, let's take another look at it:

$$Gini = \sum_{i}^{n} p_i(1 - p_i)$$

The process of randomly shuffling the data can be divided into 2 steps:

1. First, we pick one datapoint at random,
2. Then we place it in one of the bins at random.

We can see those two steps expressed mathematically in the formula:

1. p_i represents the probability that we pick a datapoint of the i-th class at random

2. $(1 - p_i)$ represents the probability that it is placed in a different class bin

So, we multiply those two probabilities and sum them for every different class present in the node. For our example node of 3 red and 7 blue samples, we can compute the gini impurity to be:

For the red class: $\frac{3}{10}(1 - \frac{3}{10}) = \frac{3}{10} \times \frac{7}{10} = \frac{21}{10}$

For the blue class: $\frac{7}{10}(1 - \frac{7}{10}) = \frac{7}{10} \times \frac{3}{10} = \frac{21}{10}$

And, summing those, we get $Gini = \frac{42}{10}$

In this manner, we can obtain the gini metric of both child nodes of a split. By combining them (based on the number of samples in each one), we can judge whether the split is good or not.

4.2 Information gain (entropy)

This is another metric to measure how good a certain split is. It is often called **entropy** because of a very similar metric in information theory. Entropy is a metric that measures how much information there is in a set. Its formula is:

$$Entropy = -\sum_{i}^{n} p_i \log_2 p_i$$

Information gain’s job is to compute the entropy (information content) in the child nodes and subtract it from the entropy of the parent in order to find how much information can be gained by making the split.

The tree generating algorithms using this metric try to maximize it (maximize the information contained in the tree). They will continue to run attempts of splitting the data until the information gain is 0 and no more information can be squeezed out of the data.

4.3 Which metric to use?

In theory, gini impurity favors bigger partitions – or distributions – whereas the information gain favors smaller ones. However, in practice, there is not much difference between the two. Researchers estimated that the metric we use matters in only 2% of the cases. In the rest, the decision whether to use gini or entropy won’t influence your results.

In fact, since information gain needs logarithms to be calculated, it turns out to be a bit more computationally expensive. That’s why most implementations will default to gini.

5 Pruning

In this section, we outline the technique of pruning in the context of decision trees.

Now, even though decision trees are a relatively simple model, they have a tendency to **overfit**. A lot. This is expressed in the tree having way too many nodes and splits, going on forever. Here is an example of an overfitted tree:

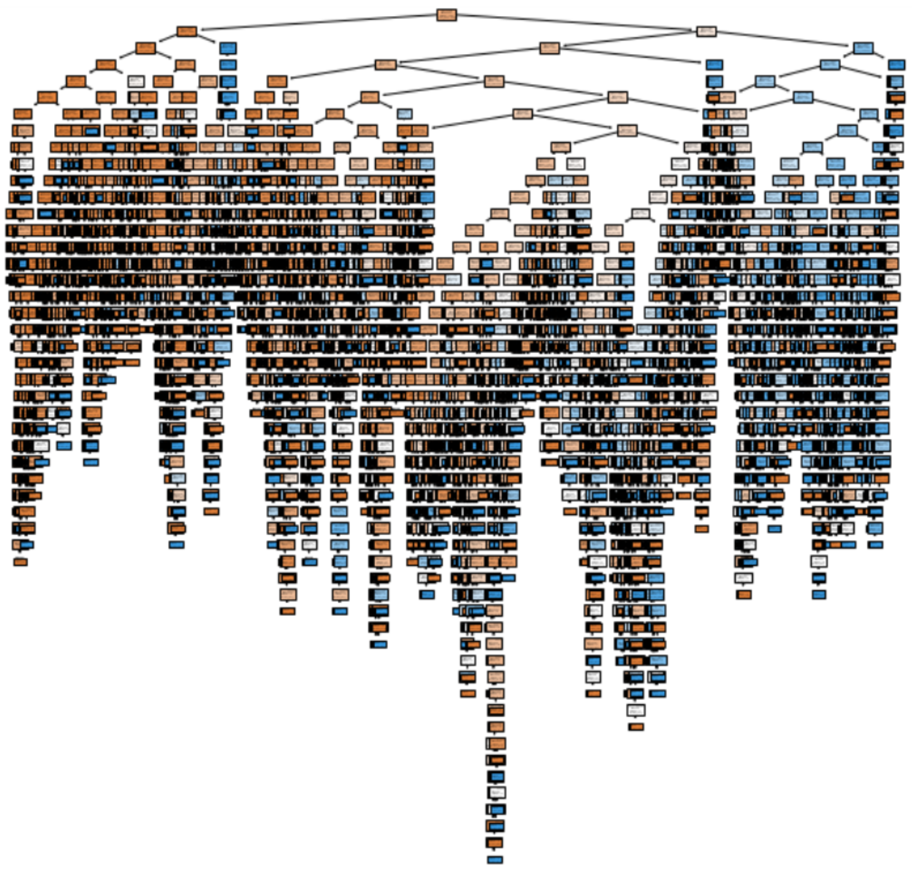


Figure 9: A heavily overfitted decision tree. There are extremely many different nodes.

We can see that this is overly complicated tree. This not only reduces the performance of the model itself, but also negates one of the main advantages of decision trees – being able to be easily visualized and understood.

Luckily, it is not all gloom and doom. There is a technique to deal with this overfitting and it's called **pruning**. As an example, here his how the same tree looks after pruning:

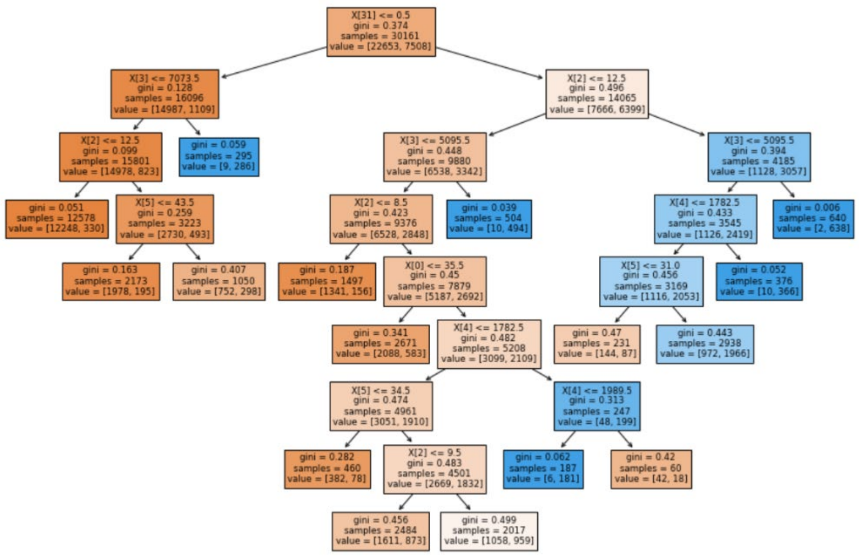

X[31] <= 0.5 gini = 0.374 samples = 30161 value = [22653, 7508]
X[3] <= 7073.5 gini = 0.128 samples = 16096 value = [14987, 1109]
X[2] <= 12.5 gini = 0.496 samples = 14065 value = [7666, 6399]
X[2] <= 12.5 gini = 0.099 samples = 15801 value = [14978, 823]
gini = 0.059 samples = 295 value = [9, 286]
X[3] <= 5095.5 gini = 0.448 samples = 9880 value = [6538, 3342]
X[3] <= 5095.5 gini = 0.394 samples = 4185 value = [1128, 3057]
gini = 0.051 samples = 12578 value = [12248, 330]
X[5] <= 43.5 gini = 0.259 samples = 3223 value = [2730, 493]
X[2] <= 8.5 gini = 0.423 samples = 9376 value = [6528, 2848]
gini = 0.039 samples = 504 value = [10, 494]
X[4] <= 1782.5 gini = 0.433 samples = 3545 value = [1126, 2419]
gini = 0.006 samples = 640 value = [2, 638]
gini = 0.163 samples = 2173 value = [1978, 195]
gini = 0.407 samples = 1050 value = [752, 298]
gini = 0.187 samples = 1497 value = [1341, 156]
X[0] <= 35.5 gini = 0.45 samples = 7879 value = [5187, 2692]
X[5] <= 31.0 gini = 0.456 samples = 3169 value = [1116, 2053]
gini = 0.052 samples = 376 value = [10, 366]
gini = 0.341 samples = 2671 value = [2088, 583]
X[4] <= 1782.5 gini = 0.482 samples = 5208 value = [3099, 2109]
gini = 0.47 samples = 231 value = [144, 87]
gini = 0.443 samples = 2938 value = [972, 1966]
X[5] <= 34.5 gini = 0.474 samples = 4961 value = [3051, 1910]
X[4] <= 1989.5 gini = 0.313 samples = 247 value = [48, 199]
gini = 0.282 samples = 460 value = [382, 78]
X[2] <= 9.5 gini = 0.483 samples = 4501 value = [2669, 1832]
gini = 0.062 samples = 187 value = [6, 181]
gini = 0.42 samples = 60 value = [42, 18]
gini = 0.456 samples = 2484 value = [1611, 873]
gini = 0.499 samples = 2017 value = [1058, 959]

Figure 10: The same tree as the one above, after pruning. It is much simpler and has 5% better accuracy.

Now this is a much better-looking tree. It even exhibits 5% better accuracy. So, how does pruning work?

It is exactly what it says on the tin. Think of how you trim bushes and plants – this is practically the same thing. In essence, pruning is a technique that removes parts of the tree that are not necessary for the final classification. It reduces the complexity of the final classifier, and hence improves predictive accuracy by the reduction of overfitting.

Pruning processes can be divided into two types – pre-pruning and post-pruning. Pre-pruning is done during the training process itself, while post-pruning is done after the tree is already generated. In practice **post-pruning** is the way **more popular method**.

In terms of pruning algorithms, there are many. But here are a couple:

Reduced error pruning

One of the simplest forms of pruning is reduced error pruning. Starting at the leaves, each node is replaced with its most popular class. If the prediction accuracy is not affected, then the change is kept. While somewhat naive, reduced error pruning has the advantage of **simplicity and speed**.

Minimal cost-complexity pruning

This is arguably the most popular pruning algorithm. This algorithm is parameterized by $\alpha \geq 0$ known as the complexity parameter. The complexity parameter is used to define the cost-complexity measure, $R_\alpha(T)$ of a given tree T:

$$R_\alpha(T) = R(T) + \alpha|\tilde{T}|$$

where $|\tilde{T}|$ is the number of terminal nodes in T and $R(T)$ is traditionally defined as the total misclassification rate of the terminal nodes. Minimal cost-complexity pruning finds the subtree of T that minimizes $R_\alpha(T)$.

In simple words, the algorithm identifies the subtree with the smallest contribution, measured in the cost complexity metric, and cuts it off from the actual tree, repeating the process until the effective parameter for the whole tree is large enough.

6 From Decision Trees to Random Forests

The random forest algorithm is one of the few non-neural network models that give very high accuracy for both regression and classification tasks. It simply gives good results. And while decision trees do provide great interpretability, when it comes down to performance, they lose against random forests. In fact, unless transparency of the model is a priority, almost every data scientist and analyst will use random forests over decision trees. So, let's see what this algorithm is made of.

6.1 Ensemble learning

In essence, a random forest is the collection of many decision trees applied to the same problem. In machine learning, this is referred to as ensemble modelling. In general, ensemble methods use multiple learning algorithms to obtain better predictive performance than any of the constituent learning algorithms alone. So, in our case, the collection of decision trees as a whole unit behaves much better than any stand-alone decision tree. In short, random forests rely on the wisdom of the crowd.

The more observant among you might have noticed, though, that through the process of creating many different trees, we lose one of the important properties that decision trees had in the first place – namely, interpretability. Even though each individual tree in the collection is simple to follow, when we have hundreds of them in a single model it becomes almost impossible to grasp what's happening at a glance. That's why this algorithm is usually treated as a black box model. That's the trade-off of random forests as opposed to decision trees – we gain additional performance and accuracy, but lose the interpretability and transparency of the model.

A logical follow-up question you might have right now is: "How do we determine what the final result should be?". After all, each decision tree produces its own answer. So, we end up with hundreds of different answers. The good news is that we can deal with this problem in a very intuitive way – by using majority voting to determine the final outcome. In other words, we choose the most common result.

6.2 Bootstrapping

The purpose of the random forest algorithm is to organically decrease overfitting. That's why there are many individual decision trees, with the idea that one tree can overfit, but many will not do so in the same manner. Thus, their average would reflect the true dependence.

A crucial part in that logic is that we don't train all of the trees on the exact same dataset. But it is rarely that we have different datasets related to the same problem, nor we can split one dataset into a hundred parts. So, how do we create many different datasets out of a single one? Well, that is the technique of **bootstrapping**.

In technical language, bootstrapping works by *uniformly sampling from the original dataset with replacement*. What that means, is that it goes through the original dataset and copies data points at random to create the new set. However, the copied points still remain in the original set and, potentially, can be copied again (one can think of them being moved to the new set and then *replaced* in the original one, that's where *with replacement comes from*). Thus, the newly generated datasets contain no new data, it's the same data but some of it is repeated. You can see two examples of this below:

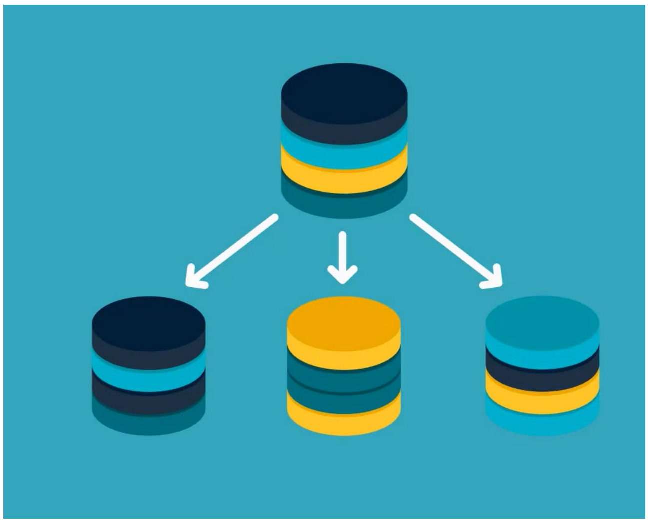
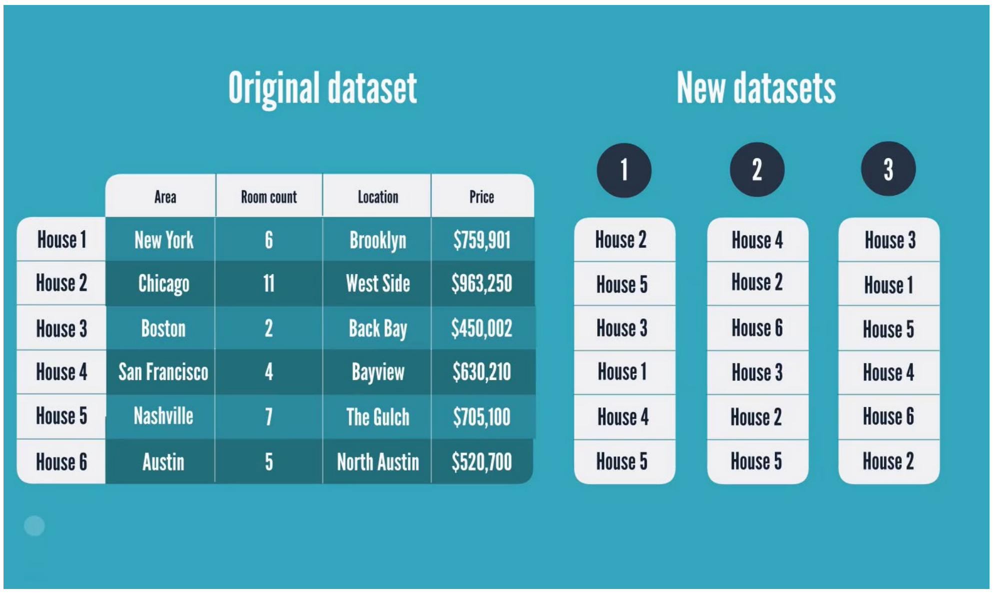


Figure 11: Schematic of datasets created through bootstrapping.



Original dataset

	Area	Room count	Location	Price
House 1	New York	6	Brooklyn	$759,901
House 2	Chicago	11	West Side	$963,250
House 3	Boston	2	Back Bay	$450,002
House 4	San Francisco	4	Bayview	$630,210
House 5	Nashville	7	The Gulch	$705,100
House 6	Austin	5	North Austin	$520,700

New datasets

1	2	3
House 2	House 4	House 3
House 5	House 2	House 1
House 3	House 6	House 5
House 1	House 3	House 4
House 4	House 2	House 6
House 5	House 5	House 2

Figure 12: New datasets generated through bootstrapping. Notice how some of the data is repeated.

These newly generated datasets are then used as the training data for the decision trees in the forest.

As a note, a dataset created in this manner that is of the same size as the original dataset is called bootstrap sample. It is expected that the amount of unique samples in it is $1 - \frac{1}{e} \approx 63\%$.

6.3 Random Forests

So far, we've discussed that we create slightly different datasets through bootstrapping, we feed them to decision trees and then collect the results and choose the final outcome through majority voting. You would be forgiven to think that this construct is the random forest. But actually, this is called **Bagged decision trees**. It stands for Bootstrap Aggregated decision trees. There is **one crucial detail** that must be satisfied in order for it to become a **random forest**.

And that is to allow each tree access to only some features, not all. That is right, in the forest, each tree can only see part of the input features. This is done to further reduce the chance of overfitting. The features to be considered are, again, chosen at random for each tree. What we can control is the size of this subset – whether we want to consider half the features, or 70% and so on.

This is, in essence, the random forest algorithm. All of the steps outlined above act as regularization to reduce the overfitting. And so, random forest rarely overfit, if not at all. This, in turn, leads to better performance of the algorithm.

7 Relevant Metrics

In this section, we introduce some of the relevant metrics that could be used to evaluate the performance of a machine learning model dealing with a classification task.

7.1 The Confusion Matrix

A confusion matrix, C, is constructed such that each entry, C_{ij}, equals the number of observations known to be in group i and predicted to be in group j.

A confusion matrix is a square 2×2, or larger, matrix showing the number of (in)correctly predicted samples from each class.

Consider a classification problem where each sample in a dataset belongs to only one of two classes. We denote these two classes by 0 and 1 and, for the time being, define 1 to be the *positive* class. This would result in the confusion matrix from Figure .

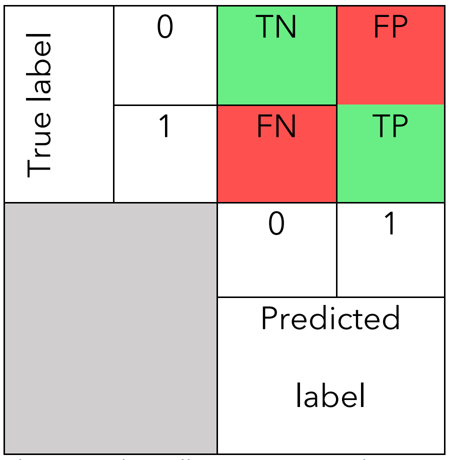
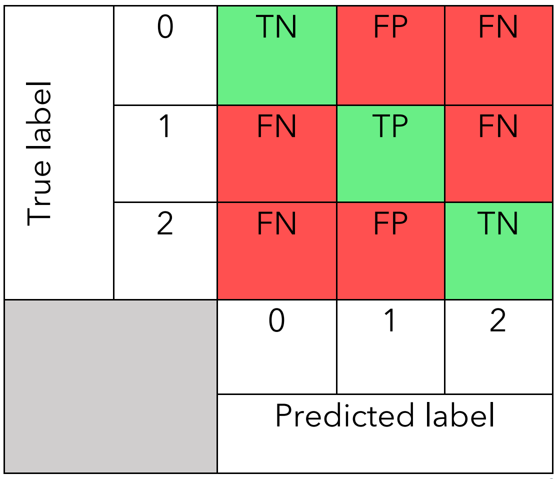


Figure 11: A 2×2 confusion matrix denoting the cells representing the true and false positives and negatives. Here, class 1 is defined as the positive one.

The matrix consists of the following cells:

- Top-left cell – **true negatives** (TN). This is the number of samples whose *true* class is 0 and the model has correctly classified them as such.
- Top-right cell – **false positives** (FP). This is the number of samples whose *true* class is 0 but have been incorrectly classified as 1s.
- Bottom-left cell – **false negatives** (FN). This is the number of samples whose *true* class is 1 but have been incorrectly classified as 0s.
- Bottom-right cell – **true positives** (TP). This is the number of samples whose *true* class is 1 and the model has correctly classified them as such.

Consider now a classification problem where each sample in a dataset belongs to one of three classes, 0, 1, or 2, with class 1 again defined as the *positive* class. This makes classes 0 and 2 *negative*. The confusion matrix would then look like the one in Figure .



True label				
	0	TN	FP	FN
	1	FN	TP	FN
	2	FN	FP	TN
		0	1	2
		Predicted label		

Figure 12: A 3 × 3 confusion matrix denoting the cells representing the true and false positives and negatives. Here, class 1 is defined as the positive one.

Making use of these confusion matrices, we introduce four useful metrics for evaluating the performance of a classifier.

7.2 Accuracy

The ratio between the number of all correctly predicted samples and the number of all samples.

$$\text{Accuracy} = \frac{TN + TP}{TN + FN + FP + TP}$$

7.3 Precision

The ratio between the number of true positives and the number of all samples classified as positive.

$$\text{Precision} = \frac{TP}{TP + FP}$$

7.4 Recall

The ratio between the number of true positives and the number of all samples whose true class is the positive one.

$$\text{Recall} = \frac{TP}{TP + FN}$$

7.5 F1 Score

The harmonic mean of precision and recall.

$$\text{F}_1 = \frac{2}{\frac{1}{\text{precision}} + \frac{1}{\text{recall}}}$$

The F_1 score can be thought of as putting precision and recall into a single metric. Contrary to taking the simple arithmetic mean of precision and recall, the F_1 score penalizes low values more heavily. That is to say, if either precision or recall is very low, while the other is high, the F_1 score would be significantly lower compared to the ordinary arithmetic mean.

Nikola Pulev

Email: team@365datascience.com

365√DataScience