MACHINE LEARNING

WITH RANDOM FORESTS AND DECISION TREES

A Visual Guide For Beginners

SCOTT HARTSHORN

Machine Learning With Random Forests And Decision Trees

A Visual Guide For Beginners

By Scott Hartshorn

Thank You!

Thank you for getting this book! It contains examples of how the Random Forest Machine Learning algorithm works. It is intended to give you an intuitive understanding of how Random Forests work, so you can apply it to larger problems. Additionally, since Decision Trees are a fundamental part of Random Forests, this book explains how they work. It also has a moderate amount of math when it dives into the details of some of the different aspects of Random Forests.

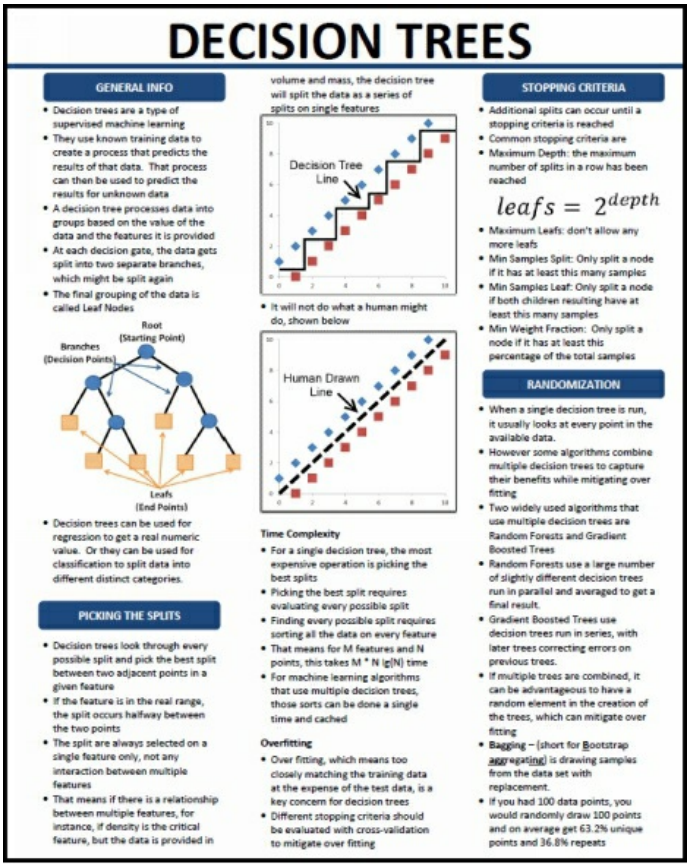
This book was mostly written to be programming language agnostic, and focus on how the machine learning works. However there is some code using the Scikit-learn module in Python 2.7

If you want to help us produce more material like this, **then please leave a positive review on Amazon.** It really does make a difference!

Your Free Gift

As a way of saying thank you for your purchase, I'm offering this free cheat sheet on Decision Trees that's exclusive to my readers.

Decision trees form the heart of Random Forests, so it is important to understand how they work. They are also a useful machine learning technique in their own right. This is a 2 page PDF document that I encourage you to print, save, and share. [You can download it by going here http://www.fairlynerdy.com/decision-trees-cheat-sheet/](http://www.fairlynerdy.com/decision-trees-cheat-sheet/)



A laminated, physical copy of this cheat sheet can also be purchased from Amazon here http://geni.us/DecisionTrees (may not be available in all countries)

Random Forest Overview

Random Forests are one type of machine learning algorithm. They are typically used to categorize something based on other data that you have. For instance, you might want to categorize animal types based on their size, weight, and appearance, or you might want to categorize a disease based on a person's symptoms. The purpose of this book is to help you understand how Random Forests work, as well as the different options that you have when using them to analyze a problem.

What This Book Is & What It Isn't

There are two different things to know to be really good at using Random Forests, or any other kind of machine learning. The first is how to set up the code, i.e. how to manipulate the data, how to generate the plots, what software to use etc. This can be considered the nuts and bolts of actually doing the job.

The second thing to know is how the Random Forest is actually working at a conceptual level. Why is it doing what it is doing, and what other options are there? If setting up the code can be considered knowing what buttons to push to make the program work, then this would be knowing why to push those buttons, and knowing when to push other buttons.

This book is focused on understanding Random Forests at the conceptual level. Knowing how they work, why they work the way that they do, and what options are available to improve results. It covers how Random Forests work in an intuitive way, and also explains the equations behind many of the functions, but it only has a small amount of actual code (in python). There is already a great wealth of free information about the nuts and bolts of exactly what to do. It can be found at places such as

- http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomFores
- https://www.kaggle.com/c/titanic/details/getting-started-with-random-forests

The following pages contain a number of diagrams, excel tables, and python programs, as well as a data set used to illustrate how Random Forests work. You can download all of these examples, for Free, here

http://www.fairlynerdy.com/random-forest-examples/

Table of Contents

Machine Learning – What Is It?

Machine learning is a deep and varied field, and this book only covers a small niche in it. But at a high level, a lot of different types of machine learning are intended to do the same thing: start with examples of something that you know, and use those to develop a pattern so you can recognize those characteristics in other data that you don't know as much about.

For example, say that you were an oil exploration company, and you wanted to make sure that in the future you only drilled wells that actually had oil in them. In the past, you might have drilled 1,000 wells, and found oil in 200 of them. For each of the 1,000 wells, you have a plethora of information that you surveyed before drilling the well. Your objective using machine learning would be to train a machine learning algorithm using all the data that you obtain before drilling a well, as well as the final result of the well after it was drilled. The algorithm would try to find correlations in the data that you have before drilling the well so that next time you can better predict if that location will have oil in it before you drill.

Random Forest is one type of machine learning algorithm. However, there are a plethora of different techniques that are applicable to different types of problems, including neural nets, support vector machines, decision trees, clustering, and others. The details of how algorithms work differ between different algorithms, but most of them are used in similar ways.

The Basics of Most Machine Learning

1. Start with a set of data that you know the answer to
2. Train your machine learning algorithm on that data set, often known as the training set
3. Get a set of data that you want to know the answer to, often known as the test set
4. Pass that data through your trained algorithm and find the result

Typically how #2 is done is what makes machine learning algorithms different from each other.

A basic example of machine learning a Random Forest in python is below and is more or less as simple as you can set up a Random Forest problem. The example shown below uses the iris data set, which is data that

distinguishes types of the iris flower based on the length and width of different parts of the flower. This example uses the iris dataset because it is widely available, and in fact, it is included in python sklearn. Many other problems in this book use a fruit dataset specifically generated for these examples. The fruit dataset was used for other problems instead of the iris data since types of fruit is more intuitive to most people, and also the plots ended up looking better.

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier

# Set the random seed so that the results are always the same
np.random.seed(12345)

# Load the Iris dataset, which is already included in sklearn
iris = load_iris()
X = iris.data
y = iris.target

model = RandomForestClassifier()

# Train the model
clf = model.fit(X, y)
```

The code shown in the book, and the code used to generate many of the examples in this book, have been adapted from the plot_forest_iris.py example program that can be found here: http://scikit-learn.org/stable/auto_examples/ensemble/plot_forest_iris.html under the BSD 3 clause license

What Is A Random Forest & How Does It Work

Random Forests are a way to do machine learning so that you can use the results from data that you have to predict the results of something that you don't know. The Random Forest is a bit of a Swiss Army Knife of machine learning algorithms. It can be applied to a wide range of problems, and be fairly good at all of them. However, it might not be as good as a specialized algorithm for any given specific problem.

Random Forests are simply a collection of Decision Trees that have been generated using a random subset of data. The name "Random Forest" comes from combining the randomness that is used to pick the subset of data with having a bunch of decision trees, hence a forest.

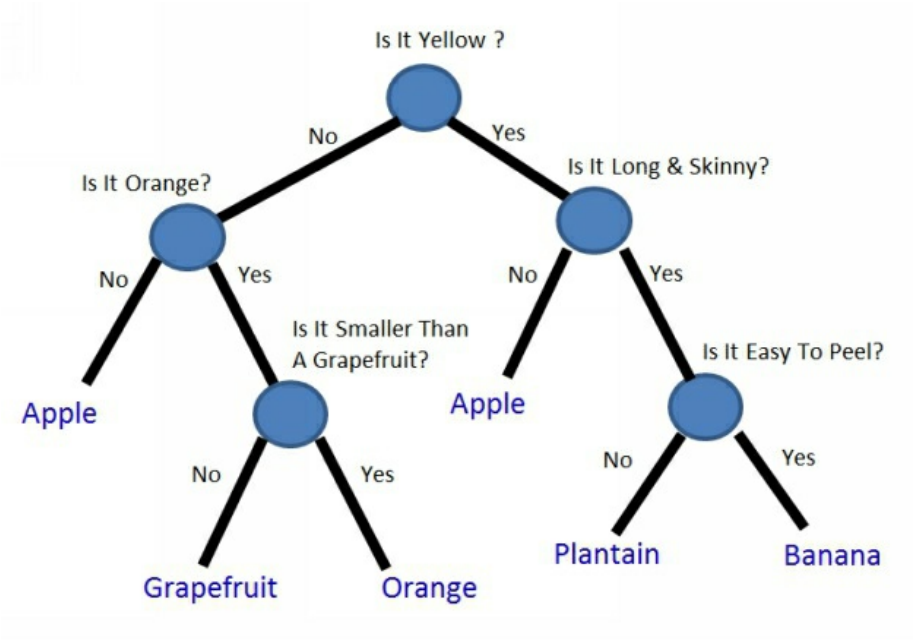
In order to understand a Random Forest, you have to understand a Decision Tree. A Decision Tree is simply a step by step process to go through to decide a category something belongs to. For example, let's say that you had a basket of fruit in front of you, and you were trying to teach someone who had never seen these types of fruit before how to tell them apart. How could you do it?

One way would just be to show the person a bunch of the fruit and name them. Hold up a banana and say "This is a banana" or "This is an apple". That method would work given enough time and enough examples, and it is basically how we usually teach children what different things are.

A different way would be to make a flow chart of questions to go through to determine what type of fruit something is. Your flow chart might have questions like

- Is it yellow?
- If so, is it long and skinny?
- If so, is it easy to peel?
- Then it is a banana

This is effectively what a decision tree is. A chart for this simple decision tree is shown below



To use the decision tree, you start at the top and begin hitting each decision in series. At each point, you need to make a choice on which way to go between exactly two options. Eventually, you reach the bottom and have a decision as to the outcome, in this case, what type of fruit something is. If your decision tree is good, you can now pick up an unknown piece of fruit and follow the flow chart to classify it. If your decision tree is bad, you can go down the wrong path and put something in the wrong category. For instance, if you didn't know yellow apples existed when you built your decision tree, you might have assumed that all yellow fruit are either bananas or plantains.

A Random Forest is made up of a number of decision trees, each of which has been generated slightly differently. In order to generate those decision trees, you need a starting data set. That starting data set needs to have both features and results. Results are the final answer that you get when you are trying to categorize something. In the example above, the results would be if the fruit was an apple, orange, grapefruit, banana, or plantain. You need a set of data where you know the results to start with, so you can use that to generate the decision tree.

The other important part of your starting data set are the features. Features are information about the item that you can use to distinguish different results from each other. In the example above, we had features such as the color, the shape, the size, and how easy it was to peel. When you pass the data set with

both features and results into the Random Forest generation algorithm, it will examine the features and determine which ones to use to generate the best decision tree. Exactly how the Random Forest algorithm does that is discussed later in this book.

One important thing to know about features is that whatever features you use to train the Random Forest model, you need to use the same features on your data when you use it. For instance, if we had generated the model with the size, shape, color, and ease of peeling features like the model above, if we then have a piece of fruit and we want to determine what kind of fruit it is, we have to know those features. If we got to the first question, and we didn't know if our fruit was yellow or not, the decision tree would be useless.

Note, it is possible that the decision trees that get generated will not use all of the features that are provided. For instance, if you pass 20 features into the Random Forest algorithm, it might only use 16 of those features in its classification. However, for most actual implementations of Random Forests / Decision Trees, when you actually want to use it to classify new data, you would still need to pass all of the same features into the algorithm that you used to generate the trees. This is because even if the algorithm only used 16 of the 20 features (for instance), it likely identified which features it was used by their location in the array. If you drop features, even ones that weren't used, most implementations will break.

Getting Started With An Example

The easiest way to start to get an intuitive understanding of how Random Forests work is to start with some examples. For this example, we are going to use different pieces of fruit, including the length, width, and color of the fruit, and see how well a Decision Tree & a Random Forest do at determining what type of fruit each piece is. Note in order to make examples that are easy to plot, this data is not exactly the same as the decision tree chart that was shown above. This dataset has different slight different features, and also only has 4 types of fruit (dropping the plantain).

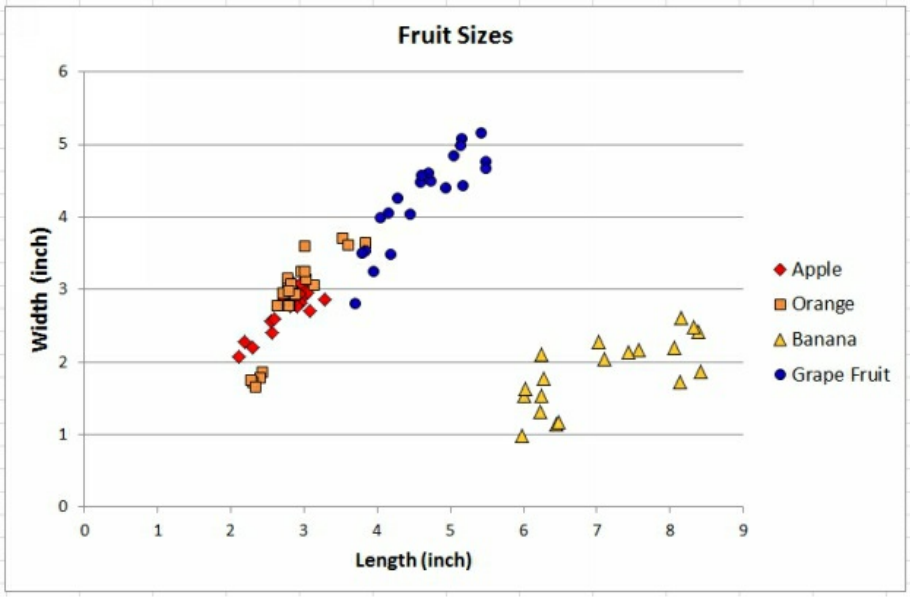
These are the different types of fruit in this example

- Apples
- Oranges
- Bananas
- Grapefruit

For colors, we assume that the fruit can be these different colors

- Apples Red, Green, or Yellow
- Oranges Orange
- Bananas Yellow or Green
- Grapefruit Orange or Yellow

For sizes, I measured some fruit of different types and came out with the sizes plotted below



In the plot, Apples are red diamonds, Oranges are orange squares, Bananas are yellow triangles and Grapefruit are blue circles. (The color choices for oranges, apples, and bananas are obvious, grapefruit gets stuck with an odd color because orange and yellow were already taken)

There are 88 total pieces of data, and this many items of each type were measured

- Apples 24
- Oranges 24
- Bananas 19
- Grapefruit 21

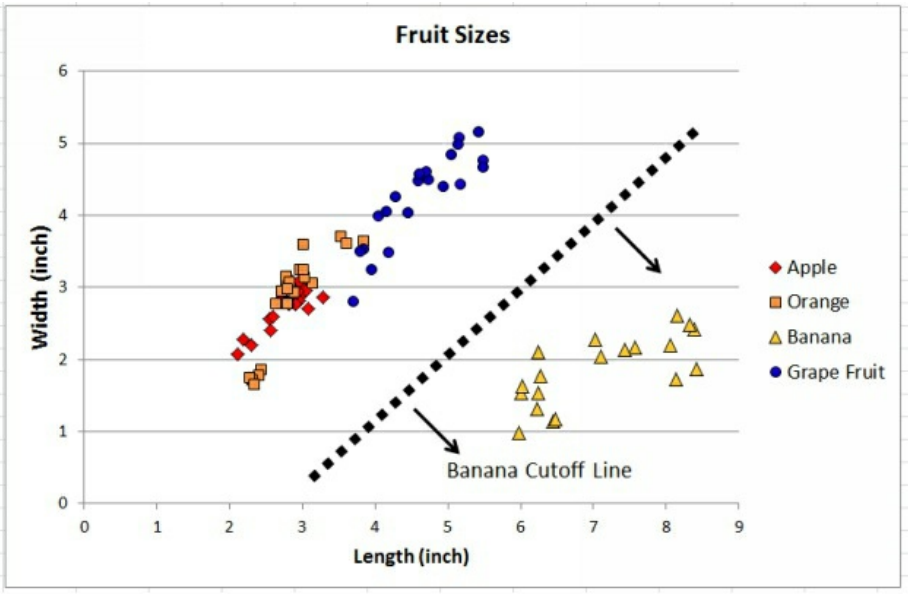
What Random Forests, Decision Trees, and other types of machine learning algorithms are most useful for are taking higher dimensional data, or a large quantity of data, and making sense of it. In this case, we only have 3 dimensions to the data, i.e. 3 different variables, the color, length, and width, and only a small quantity of data. Therefore we expect that a person trying to classify these fruits would be just as good, if not better, than the machine learning algorithm.

Which is great! That means we will be able to completely understand what the algorithm is doing for this small set of data. Then if you want to extend that to a higher order dataset with more variables, i.e. one that is difficult to plot and classify manually, you will know what the algorithm is doing for that larger case.

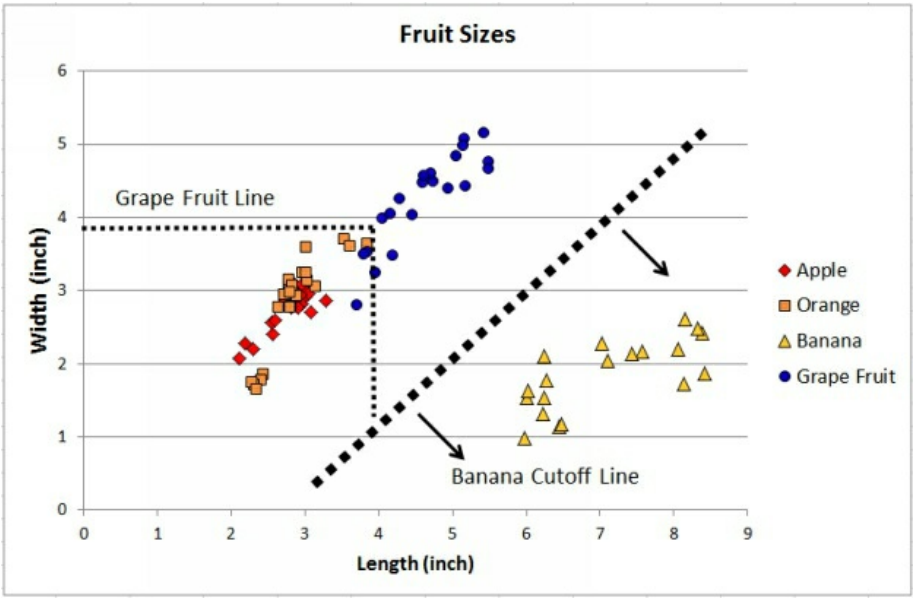
Classifying The Fruit Manually

If we gave this fruit classification problem to a person instead of an algorithm, how would they do it? In fact, forget about the color of the fruit, if you had to classify the fruit solely based on their sizes plotted on the chart above, how could you do it?

Well, the first thing to do is separate the bananas. Everything else is more or less round, but the bananas are long and skinny which make them the low hanging fruit of this problem. So we can draw a line between the bananas and the round fruit and say that anything on one side is a banana, and anything on the other side is the one of the other fruit, which we will figure out later



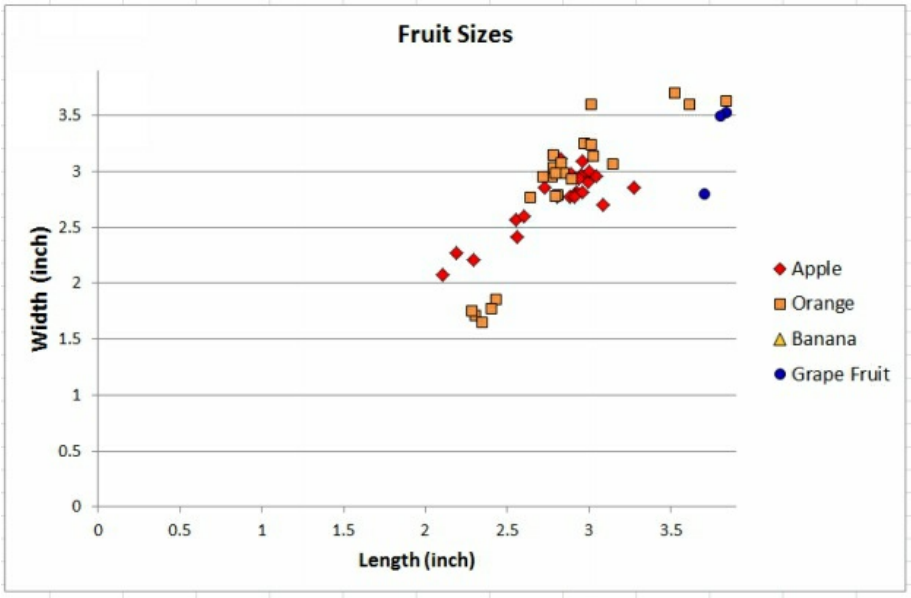
With a single line, we've classified one category. Now that the bananas are separated out, how about the rest of fruit? Well, they are more challenging, but we can still make some judgment calls that are better than simply guessing between the three remaining fruit categories. To start with, most of the grapefruit appear to be larger than either the apples or the oranges, so we can draw a line and say that anything larger than 3.9 inches in either length or width is a grapefruit



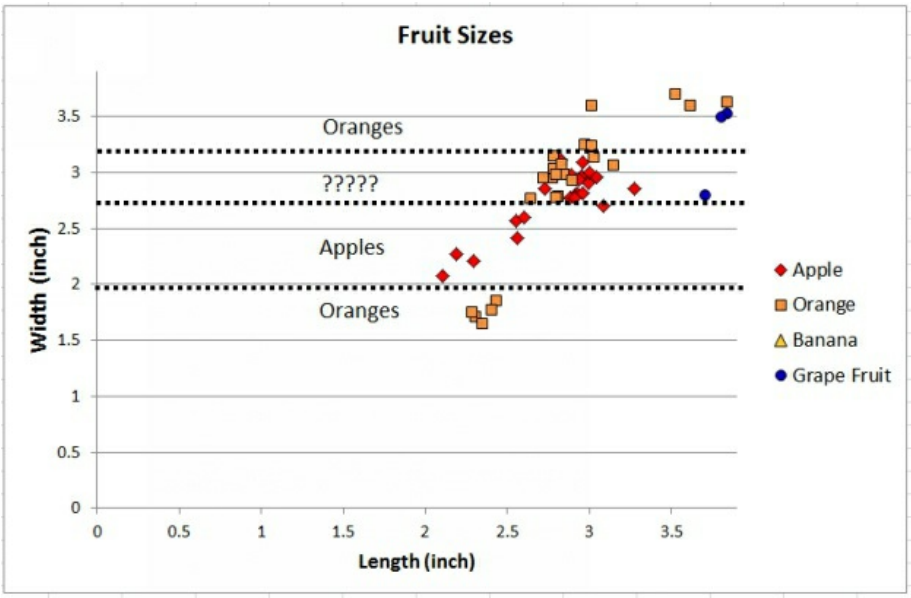
What we see is that we were able to split off most of the Grapefruit, but not all of them. There are a few Grapefruit that are smaller than our cutoff lines. If we had made our cutoff line smaller, then we would have started to pick up oranges and classify them as Grapefruit, so it might not be possible to always tell the difference between oranges and grapefruit just by this length and width chart.

Intuitively that makes sense. Even in real life, it isn't always easy to tell the difference between a large orange and a small grapefruit. There have been a few times I have sliced into what I thought was an orange, only to discover that it was a grapefruit. So judging by size alone, it makes sense that we can't always tell the difference between the two.

So we accept that when we try to manually classify these fruit there might be some errors, and we move on. The remaining area that we have to classify is the bottom left corner, which looks like this when zoomed in.



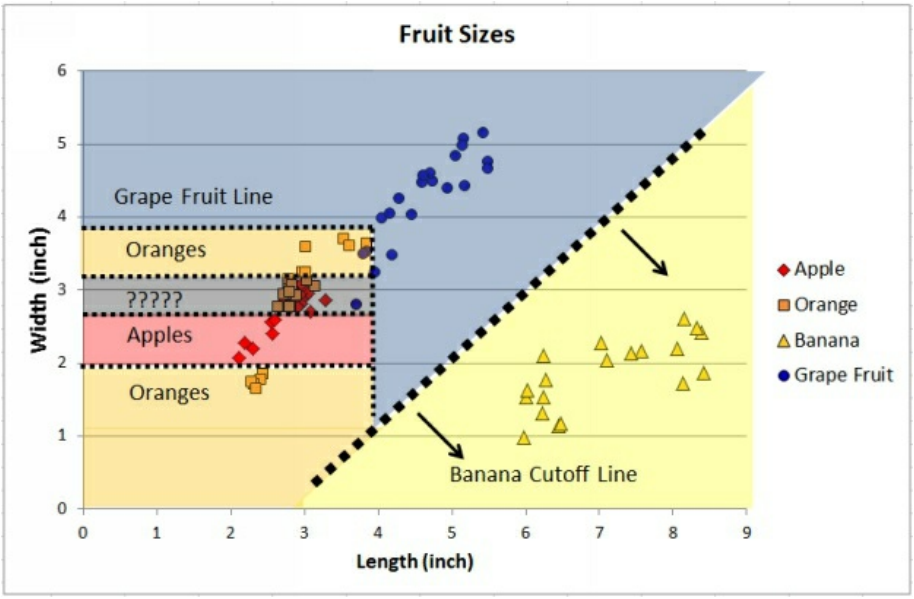
Based on this data, we might say that the really small fruit are always oranges (in fact they are the clementine oranges), the kind of small fruit are apples, and the largest fruit are oranges. Realistically, we are probably not thrilled with having to classify oranges vs apples based on their size alone, since we know they span a range of similar sizes, and we know that if we could look at their color it would be easy. None-the-less, if forced to make a decision based on this data alone we would probably draw lines like this



We were able to split off some of the oranges and apples, but are left with an area in the center where there are a ton of oranges and apples that all

overlap. For that area, we would probably throw up our hands and say that something in that size range could either be an orange or an apple, but we can't tell you which.

The end result is a plot that looks like this



Where anything that falls in the yellow zone we would call a banana, anything in the blue zone is a grapefruit, in the orange zone is an orange, in the red zone is an apple, and in the gray zone, we don't know.

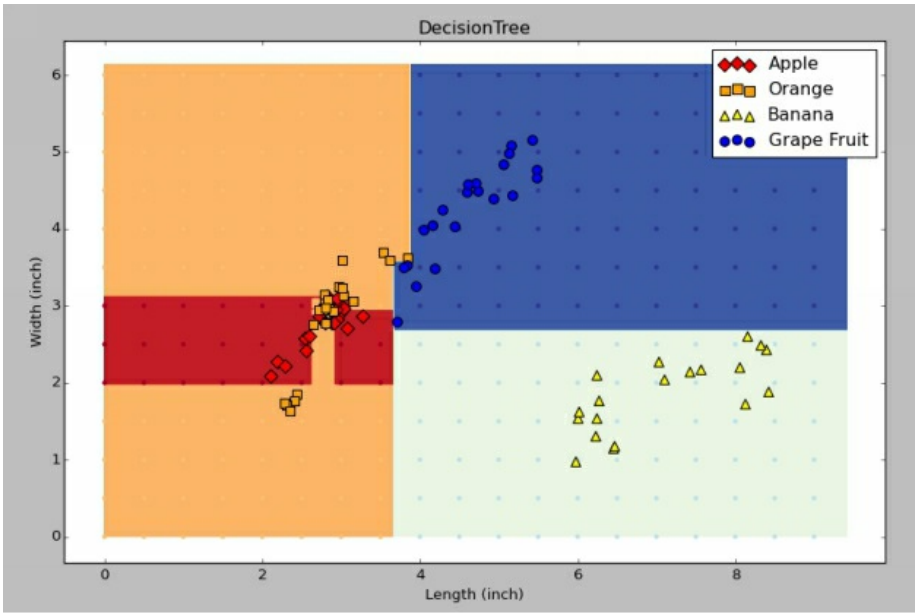
A couple of things that become obvious when we look at this chart is that there are wide swaths where we don't really have any data, and that are just getting categorized by things that are in the general vicinity. So although we forced ourselves to color the whole chart, if we get data that is significantly different than what we use to generate the chart, we might find areas of the chart that are incorrect.

Although different people might draw the lines above differently, it is certainly a reasonable way for a human to classify fruit based on size alone. This chart is basically a decision tree that we made manually. Since the point of this book is to explain how computers would do it using the Decision Tree & Random Forest algorithms, it is time to dive into different algorithmic techniques.

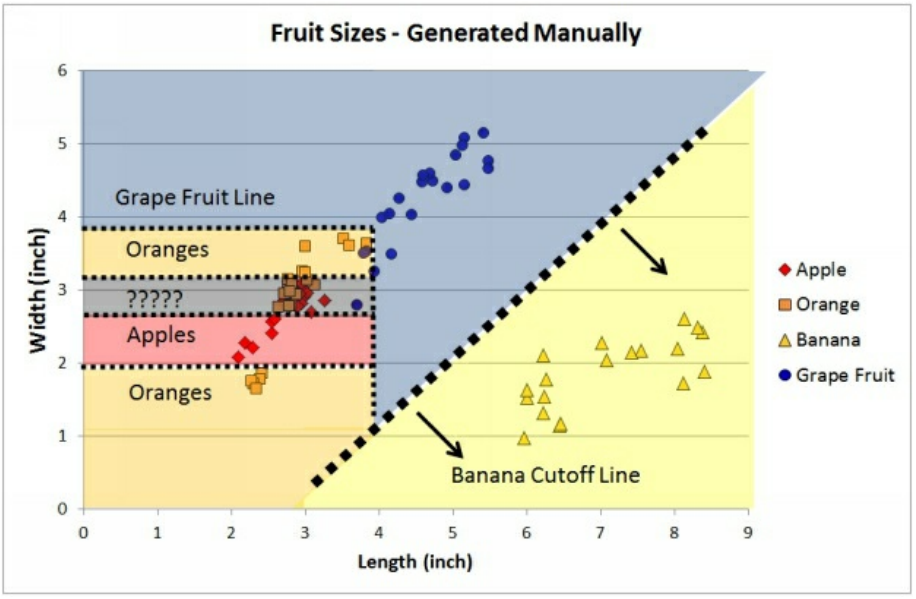
Decision Tree Example

Remember, the Random Forest algorithm is just a collection of decision trees that were each generated slightly differently. So it makes sense to start by looking at how one decision tree would classify the data. A decision tree classification is plotted below.

The plot was created by generating a decision tree and then testing thousands of length, width combinations and shading each location based on its result from the decision tree. There is also a coarser plot of a few hundred dots generated the same way. Those dots will be more meaningful in the Random Forest section. You can get the Python code that ran the classification and generated this plot here. http://www.fairlynerdy.com/random-forest-examples/

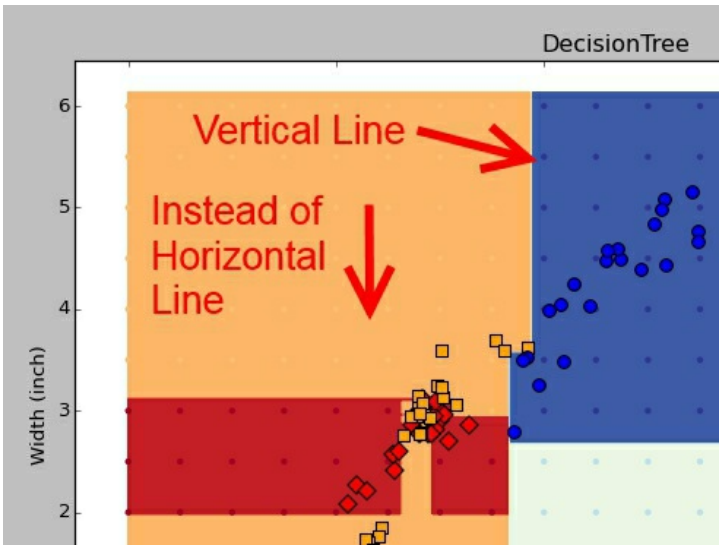


At a high level, this plot is pretty similar to the one that we generated manually

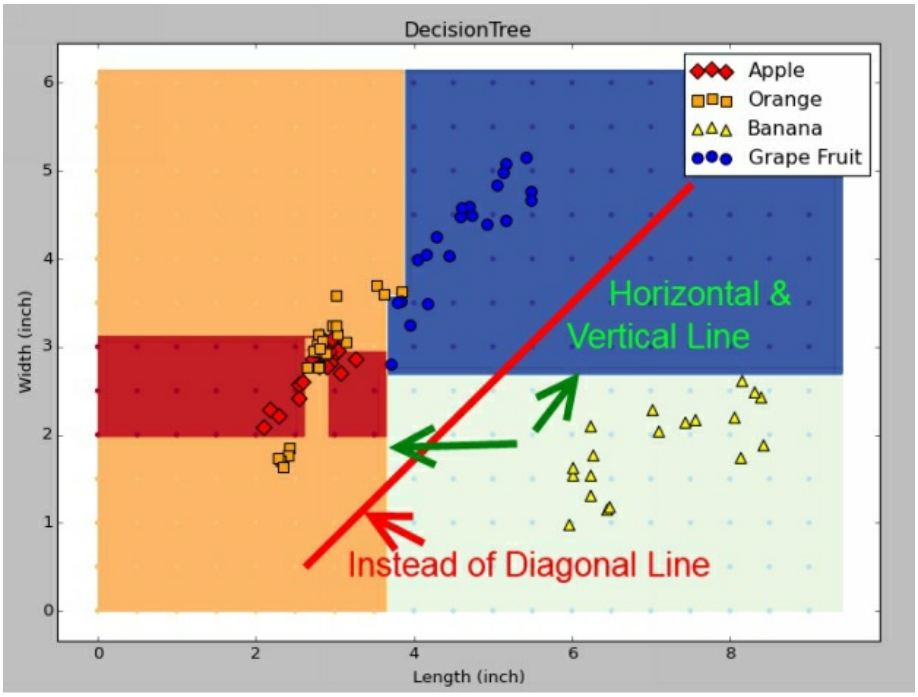


Both of them have split off bananas into the lower right corner, grapefruit into the top right, oranges in the bottom left, and apples in the bottom middle. However, there are clear differences between the plots. We can use those differences to help us understand how the decision tree algorithm differs from what we did manually.

One of the most obvious differences is how the decision tree colored the top left. It classified the area as oranges for oranges, instead of blue for grapefruit like we did. However, this difference turns out to be not very significant for gaining an understanding of how the decision tree works. It is just because the algorithm opted to split the oranges from the grapefruit with a vertical line instead of a horizontal line like we did. Using the vertical line is something that we could have chosen to do just as easily.



A more important difference is how the Decision Tree chose to split off the Bananas. We did it with a single diagonal line. The decision tree used two lines, one horizontal and one vertical.

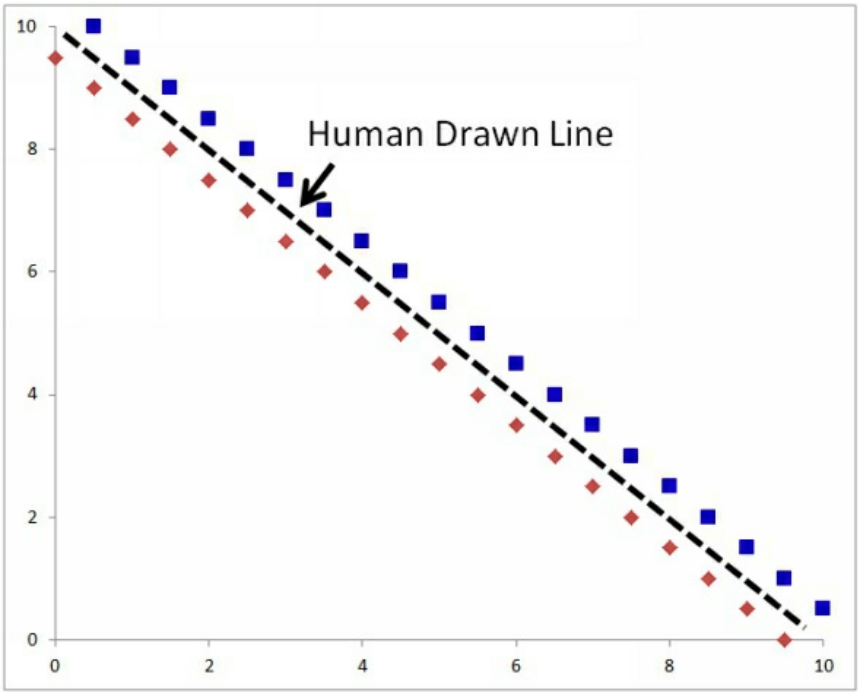


The decision tree works by picking a criteria and a threshold. The criteria specifies what to split, for instance length or width, and one single criteria is always used. The threshold specifies where to split, i.e. what value to split at.

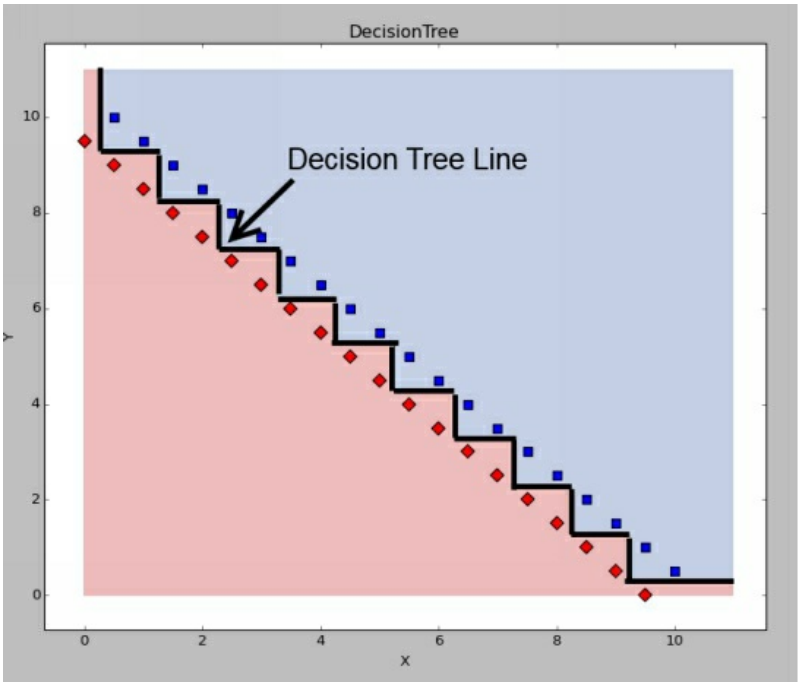
But to get a diagonal line, we would need to split on two values simultaneously. The best a decision tree can do is to split on one value, and

then the other value and repeat that process.

Take this example with made-up data



It is obvious to us that the best split between the red and blue data points is a diagonal line from top left to bottom right. But the decision tree classifier would take quite a few steps to do it, and end up with a stair step like plot, as shown below.



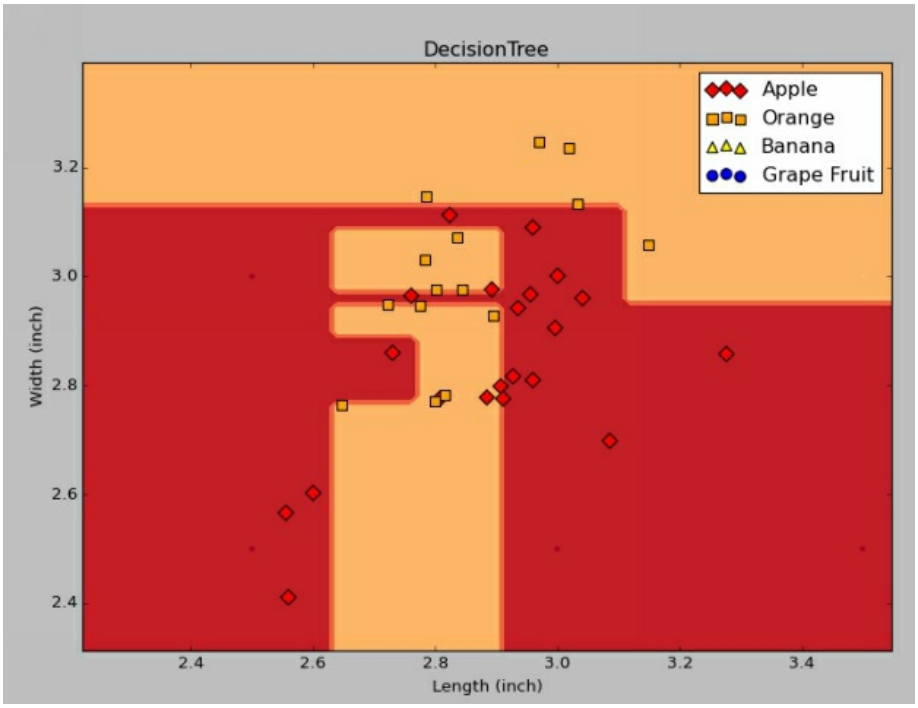
This ends up being one way to improve the results of a random forest. If you see a relationship, like a ratio, between different criteria in the data you can make it easier for the code by making the ratio its own value. The example above took 20 lines to split the red dots from the blue dots. But the data itself is very simple. It is just the expression of the equation $y = 9.5 – x$ for the red dots, and $y = 10.5 – x$ for the blue dots. If I had added the X value to the Y value for this plot, the Random Forest could have made the split in a single line, just anything with Y below 10 was red, and anything with Y above 10 was blue.

To put it a different way, if you were trying to classify objects based on if they would float or not, you might have data based on their weight and their volume. If you use a Random Forest on that data, you will get a result, but it might be complicated. However, if you edit your data and pass it a criteria where you divide weight by volume to get density, then you really have something useful.

Overfitting in Decision Trees

Going back to the fruit example, another difference between how the computer classified the data and how we would do it is the level of detail. Eventually, we got to a spot where we gave up classifying the data. We recognized that the large grouping of apples and oranges was essentially random. So trying to separate everything would not be useful.

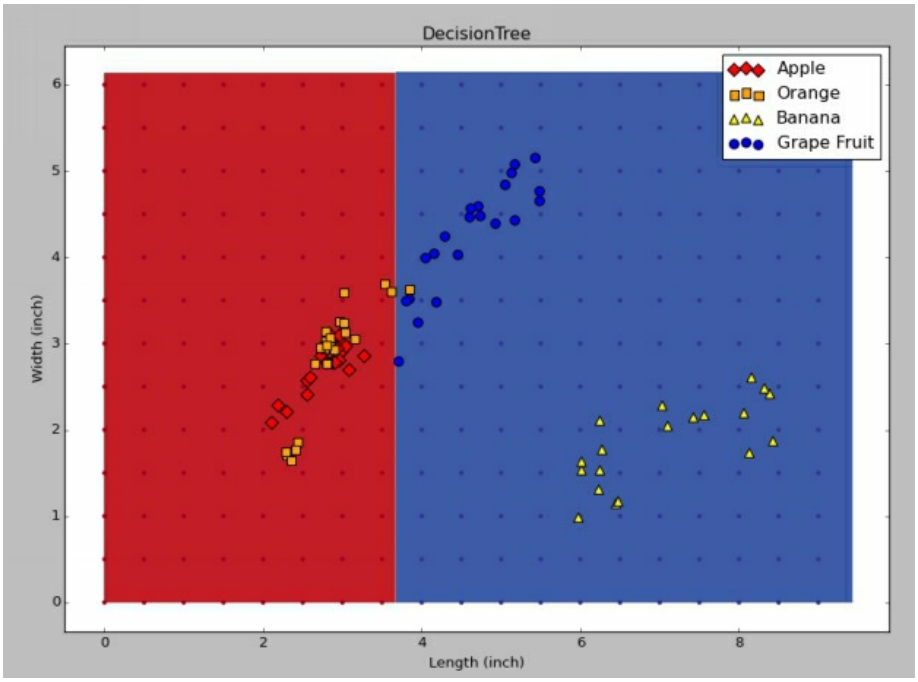
The computer never threw up its hands and stopped. By default it continues until every single piece of data is split into a category that is 100% pure. Zooming in on the decision tree, we can see this leads to incredibly fine distinctions in how an area is categorized.



This is an example of overfitting. Overfitting means that we are drawing too fine of conclusions from the data that we have. The whole purpose of making this classifier is so that when we get a new piece of fruit, we can measure the length and width and know what type of fruit it is. However, the new pieces of fruit will all be a little bit different than the data that we already have. So trying to say that a piece of fruit with a width of 2.87 inches is an apple, but one that has a width of 2.86 inches or 2.88 inches is an orange is an example of thinking that there is more precision in the data than we actually have.

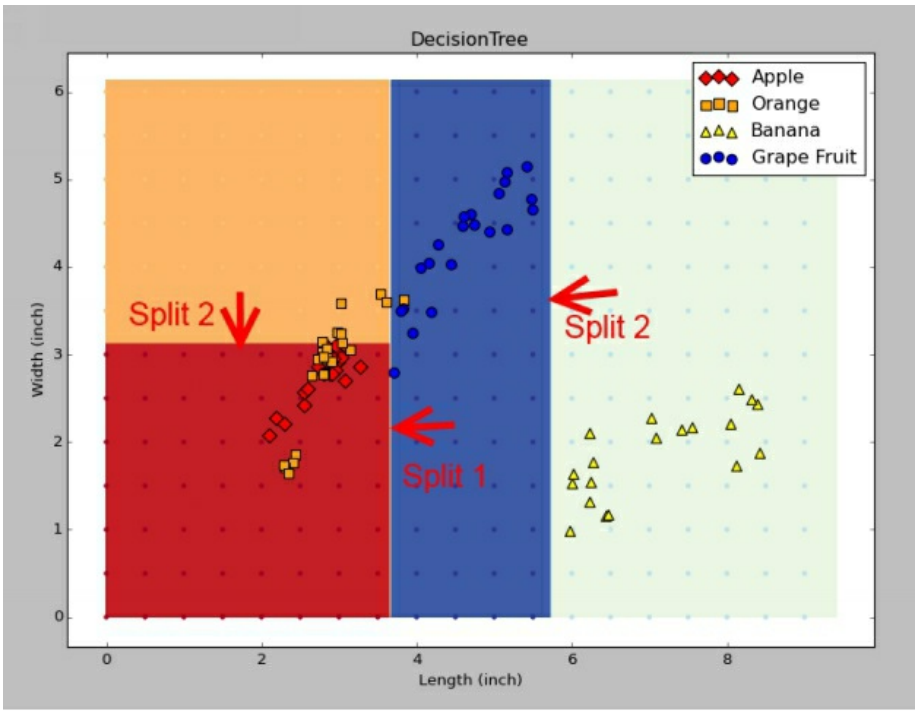
There are a couple ways to control overfitting in decision trees. One way is to limit the number of splits that the decision tree makes. So far, we have let it make as many splits as it wanted until it got every piece of data into a pure category. We could direct it to make no more than 1 split, or 2 splits, or 10 splits, or any other number to limit the number of splits.

This is an example of what the decision tree looks like if we limit it to only one split



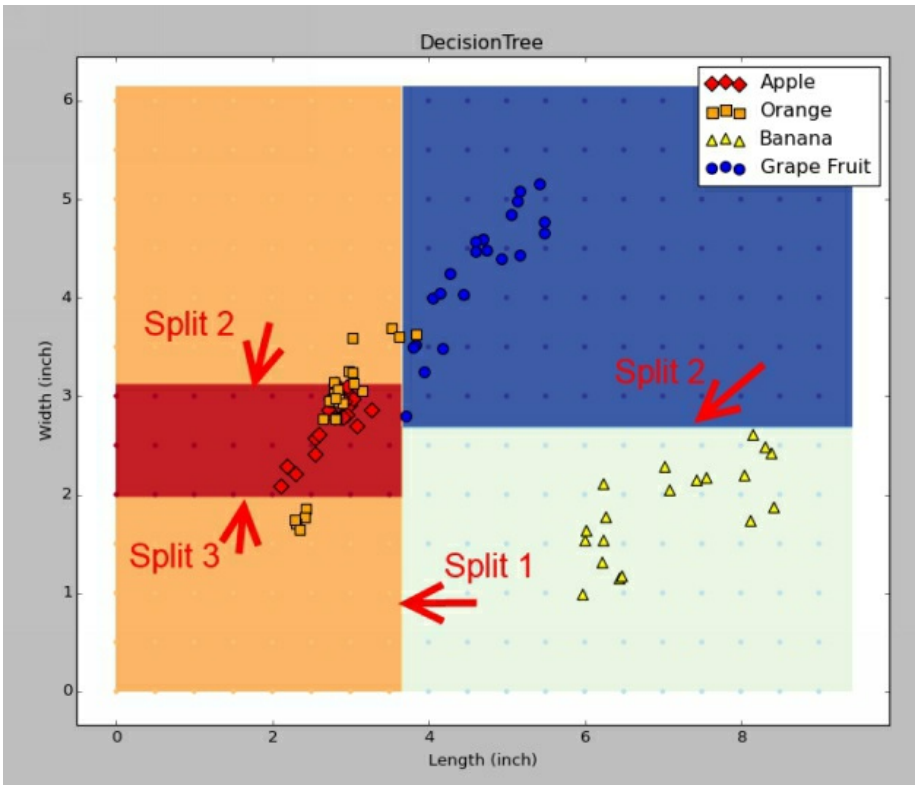
Obviously, this is not very good. But we didn't expect it to be since we can't break into 4 different categories with only one split.

Let's keep adding splits, and see how this changes. Here is the decision tree with 2 splits.



Notice that even though we specified the maximum number of splits as 2, there are actually 3 splits. This is because specifying a maximum number of splits as 2 really meant a maximum depth along any given branch was 2. So the first split made 2 new branches, and the second split made each of those into 2 branches, for a total of 4 branches.

If we allow 3 splits, the plot looks like this



Here some of the splits changed, i.e. split 2 on the right is now horizontal instead of vertical, because the Random nature of which criteria are examined when performing splits. Even though we have set a random seed, the different number of splits means the branches are analyzed in a different order, which changes the random state at different branches.

We could keep adding splits, but it becomes hard to distinguish exactly where they are all made and isn't really that useful for understanding how the decision trees work. If we added enough splits, we would get back our original plot with the overfitting. If you want to generate these yourself, you can find the code I used here, http://www.fairlynerdy.com/random-forest-examples/ and the parameter that you need to change is max_depth, an example of which is shown below

```
model = RandomForestClassifier(max_depth=3)
```

Additional Options For Limiting Overfitting in Decision Trees

The other way to limit the overfitting it to only split a branch of the decision tree if there are a certain number of data points on it. Currently, if we had a branch that had one apple, and one orange, the decision tree would split that into two branches. However, we might decide that if we don't have at least 6 pieces of data on a branch, then we shouldn't split because we might be overfitting.

These limits can be set using different parameters. In python, you could do it as shown below

```
# Only split a branch if there are at least 6 nodes in it
model = RandomForestClassifier(min_samples_split = 6)

# Only split a branch if both children will have at least
# 3 nodes on them
model = RandomForestClassifier(min_samples_leaf = 3)

# Only keep the best 20 leafs
model = RandomForestClassifier(max_leaf_nodes = 20)
```

Random Forest Example

Up until now, we have looked at Decision Trees, which are components of a Random Forest. Now it is time to look at the Random Forest itself.

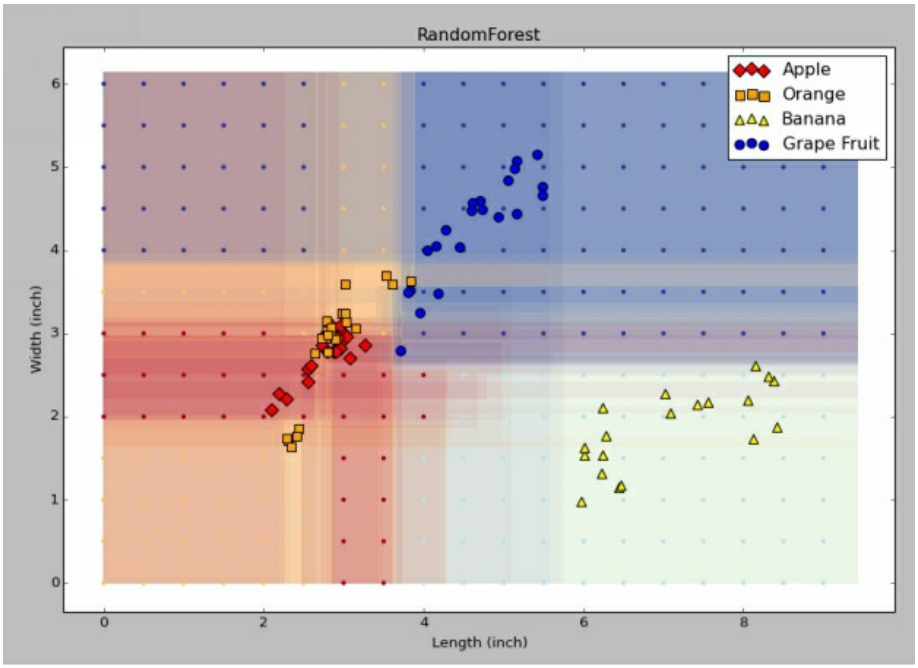
So How Is A Random Forest Different

The above example showed how a decision tree works. And while an individual decision tree is useful, it does have some limitations. As we saw, one of the most severe limitations is the tendency for decision trees to over-fit their data.

In many real-world examples, it can be challenging to classify things based on the data that is given. You can have messy data and anomalies that don't generalize to the real world. A decision tree will not smooth out those anomalies. If you are trying to tell the difference between oranges and grapefruit based on their size, a decision tree might break the data down into very small, specialized ranges that work for your data, but not for any random fruit that might come in.

Random Forests attempt to fix this problem by using multiple decision trees and averaging the results. However, just generating multiple decision trees using the same set of data over and over is not beneficial since you will just get multiple copies of the same decision tree. (Or at least very similar decision trees, depending if there is any randomness in their creation or not). So Random Forests generate their decision trees using subsets of the full data set that are randomly selected.

This is an example of the fruit data analyzed with a random forest. Just like in the decision tree example, the random forest plot below has only been trained on the fruit size. The color criteria was omitted to make it easier to plot in 2 dimensions.



The most obvious difference between the random forest plot and the decision tree plot is that the colors are not pure anymore. There are overlapping shades of different colors. What this is representing is that this random forest was generated with 16 different decision trees. Each of those 16 different decision trees was generated with a slightly different set of data. Then the results for each of the decision trees were combined.

For some areas, such as the very bottom right by the bananas or the top middle by the grapefruit, all or nearly all of the decision trees reached the same conclusion so the colors are not shaded between multiple colors. For other areas, different results were generated on different decision trees, so there are overlapping colors.

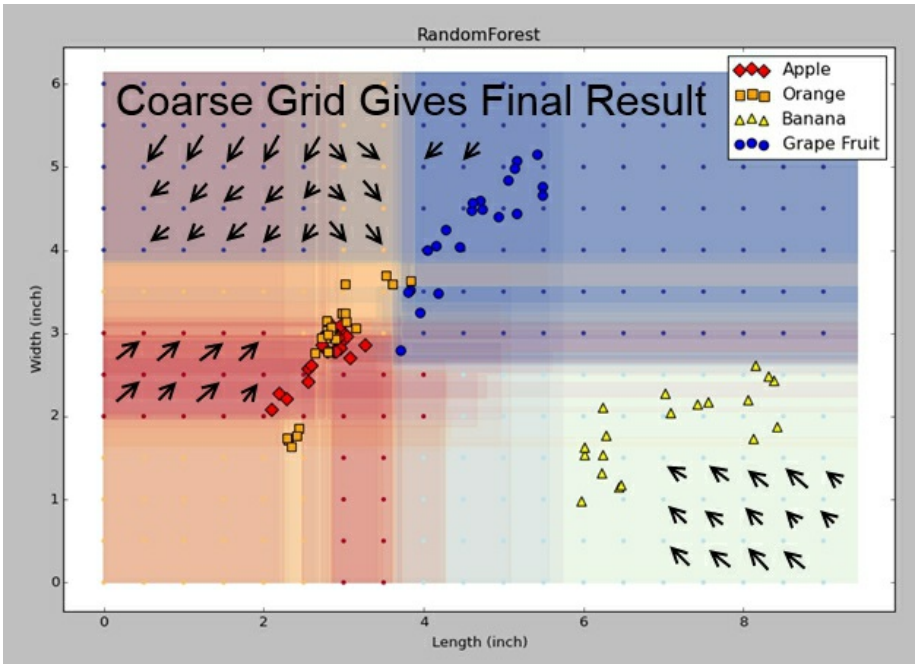
Predicting With Random Forests

So what do you do with a plot that has shades of multiple colors at the same location? If multiple decision trees are giving you different results, which result should you use? Random Forests solve this problem with voting. There are two different ways the voting can work. The first is just to count all the votes by all the decision trees and take the highest count as the solution. The other way is to count all the votes and return a result based on the ratio of the votes.

The first way where highest vote count wins is similar to how politicians are elected. The second way is analogous to 5 friends arguing over pizza toppings, and finally ordering pizza with 60% pepperoni, 40% mushrooms.

More mathematically, the second answer that could be generated is a weighting of all the results. So if 9 trees are predicting orange, and 7 trees are predicting grapefruit, the result returned would be 56.25 % orange, and 43.75 % grapefruit. Note – what is actually occurring is just an averaging of all the results of all the trees. So if you have limited the trees such that they have leafs that are not 100% pure, then even a single tree might return a result that is 60% one category and 40% another. That weighting would just get averaged in with the rest.

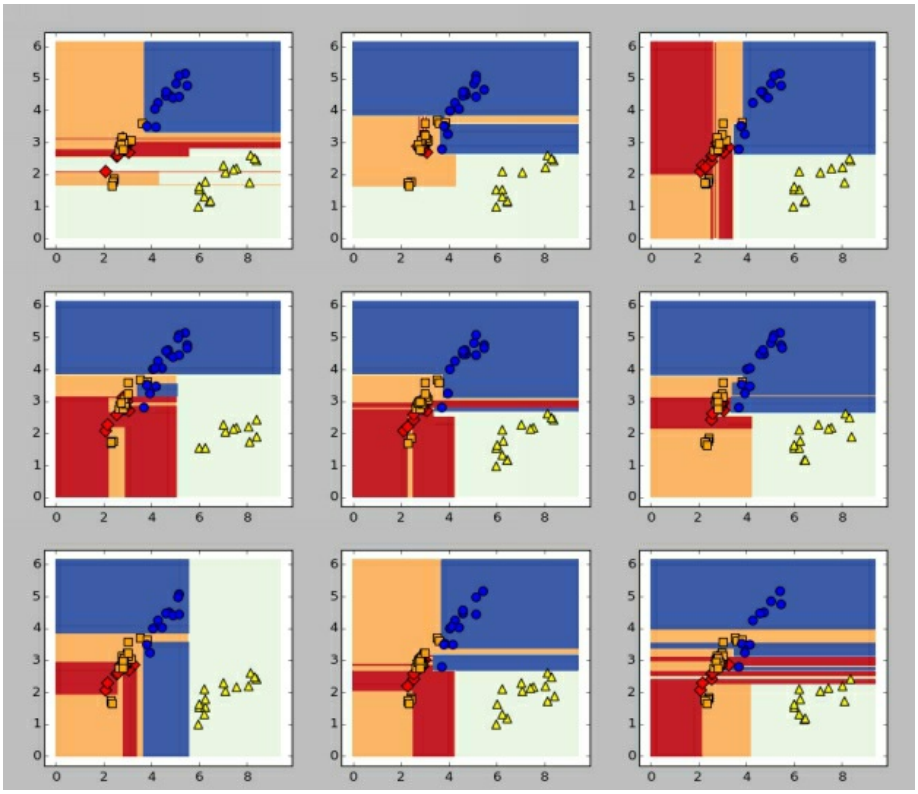
For this Random Forest chart, we have used the shading to show the results from the different decision trees, and the more coarse evenly spaced grids to show the final result from the Random Forest.



In python, if you just want the most votes for the most common category, you can use the predict(X) function. If you want to return the weightings for all the different trees, you can use the predict_proba(X) function. More details on how to use those functions can be found here.

http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier

The previous graph with the shadings shows the results of processing this fruit dataset with 16 trees in the Random Forest. The plot below shows 9 of those decision trees, each colored based on how they individually classify the data. (The plot below shows 9 decision trees instead of all 16 in order to make them large enough to view)



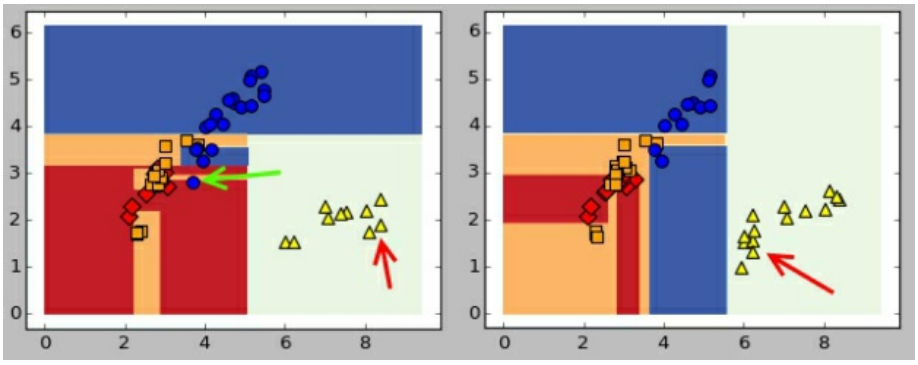
The most obvious difference between the 9 different charts is the shading. Although they are all broadly similar, there are clear differences between each of the trees. Those differences are most pronounced in the same areas that we previously had trouble categorizing. i.e. between apples and oranges, and also some areas between oranges and grapefruit. The easiest, most obvious areas are nearly all the same, such as the bottom right being bananas or the top right being grapefruit.

The differences in the shading between the different decision trees are what generate the areas of different colors in the Random Forest chart that was shown earlier. Each of the different decision trees contributes to the overall Random Forest result. Where all the decision trees agree, there is a consistent color. Where the decision trees have different results, there is shading of two or more colors.

A less obvious difference between the charts is that they don't all have the same data on them. The data on each of them is all selected from the same source, but on average they each only have 63.2% of the original dataset. This is known as Boot Strapping and is covered in the next section.

The picture below shows two of the trees, and the arrows point to some of the

data that is different between the two of them.



Because the decision trees are built with different data, they will not all be the same. This is one area of the randomness in a random forest.

The Randomness In A Random Forest

There are two ways that randomness is inserted into a Random Forest. One is based on what data is selected for each tree, and the other is based on how the criteria for the splits are chosen.

Getting Data For Each Tree: Boot Strapping

All of the Decision Trees in a Random Forest use a slightly different set of data. They might be similar, but they are not the same. The final result is based on the votes from all the decision trees. The outcome of this is that anomalies tend to get smoothed over since the data causing the anomalies will be in some of the decision trees, but not all of them, while the data that is more general will be in most if not all of the trees.

When generating each tree, that tree has a unique set of data. That set is generated from a random subset of all of the available data, with replacement. This technique is known as bootstrapping. Each of the trees uses a set of data that is the same size of the original dataset.

As an example, let's say that you are generating a tree from the set of numbers 1-10. The original data set has 10 numbers in it, so the new tree will have 10 numbers in it. The original data set has numbers only from the range 1-10, so the new tree will have numbers only from the range 1-10.

However, the original data set has all of the numbers 1-10, the new tree will probably only have some of them. For instance, a tree might be

1, 2, 2, 3, 5, 7, 7, 7, 9, 9

This was generated by picking 10 numbers out of the set 1-10, with replacement. i.e. Once a number was picked, it could be picked again.

If you generated another tree, it might be

2, 3, 3, 5, 6, 6, 7, 7, 8, 9

This tree will be similar to the first tree that was created since they use the same pool of data, but they will not be the same. For instance, this tree includes 8 in its set and omits 1. Both trees still omit 10, just by luck of the draw.

If you run this random sampling enough times, with large enough data sets,

you will find that on average 63.2% of the original data set is in each tree. Since each tree is the same size as the original data set, that means the other 36.8% is duplicates of the original dataset. In this case, since there are only 10 items in the data set, the bootstrapping would on average select 65.1% of them each tree it generated. That can be calculated as

$$1 - (1 - \frac{1}{N})^N$$

For 10 items, it would be 1-.9^10 = 65.1%. For 100 items it would be 1-.99^100 = 63.4%. For 1000 items it would be 1-.999^1000 = 63.2%. By 1000 items this equation is pretty well converged on 63.2% of the dataset being included in any given tree.

Since most Random Forests contain anywhere from a few dozen to several hundred trees, then you are likely that each piece of data is included in at least some of the trees.

Criteria Selection

The other way that a random forest adds randomness to a decision tree is deciding which feature to split the tree on. Within any given feature, the split will be located at the location which maximizes the information gain on the tree, i.e. the best location. Additionally, if the decision tree evaluates multiple features, it will pick the best location in all the features that it looks at when deciding where to make the split. So if all of the trees looked at the same features, they would be very similar.

The way that Random Forest deals with that is to not let the trees look at all of the features. At any given branch in the decision tree, only a subset of the features are available for it to classify on. Other branches, even higher or lower branches on the same tree, will have different features that they can classify on.

For example, let's say that you have a Random Forest trying to classify fruit into either pears or apples. You make the Random Forest with 2 decision trees in it, and you pass it 4 features that it can classify on

- Weight
- Size
- Color
- Shape

In many implementations of the Random Forest algorithm, it will use the square root of the number of features as the maximum features that it will look on any given branch. Using fewer than all of the available features is another way to add randomness to the results (and avoid overfitting) as well as reduce the time required to create the trees. The number of features retained for each tree is an adjustable parameter, and the best value can vary based on the date set. In this case, we have 4 features, and if we assume that we are keeping the square root of the number of features for each tree, each decision tree will use the best of the 2 randomly selected features available.

The first tree for the first choice may have to pick the best classifier between Color & Size. The best choice might vary by the dataset, but pears and apples are a similar size, so it might pick color. The next decision branch gets to choose between two features independent of what any previous branches evaluated. This might be color and weight. It will pick the best split

possible with those criteria. If there are absolutely no improvements available using the criteria options it has, it will continue evaluating additional criteria to find a useful split.

How Many Trees In A Random Forest?

A Random Forest is made up of multiple decision trees, but exactly how many decision trees is that? That is something that you need to decide when you are generating your Random Forest. Most software lets you control the number of trees in the Random Forest with a parameter. This is how you would control it in python, setting the number of trees to be 100 instead of the default of 10.

```
model = RandomForestClassifier(n_estimators = 100)
```

To start with, more trees are usually better because they will do more to smooth out abnormalities in the data. But that is true only up to a point. This is an area of diminishing returns, where each additional tree will have less benefit than the one before it. Eventually, the benefit will plateau, and more trees will not help very much.

Conversely, the amount of time it takes your program to run will likely scale linearly with the number of trees that you have. Each one will need to be generated individually, and the test data will need to go through each one individually. So if you have 1,000 trees instead of 100, that portion of your program will take 10 times longer to run.

The decision on how many trees to have in the Random Forest becomes a tradeoff dependent on your problem and your computing resources. Going from 10 trees to 50 trees might improve your results significantly, and not add very much time. Going from 1,000 trees to 50,000 trees might add substantial time without improving your results very much. A good plan is to start with a few trees and scale up that number using your cross-validation results to determine when the benefit is no longer worth the additional runtime. Using 100 trees in the Random Forest is a good place to start.

Getting improvements by dramatically increasing the number of trees in your Random Forest is often one of the things that are worth doing last since you know that more trees will almost always be at least some benefit. It is usually better to study the other parameters you can tune, and study improvements you can make to your data with a smaller number of trees that run quickly, and save increasing the number of trees until the end.

Out Of Bag Errors

Out of Bag error is an important concept in Random Forests. Important enough that "Out of Bag" often gets its own acronym O.O.B. The reason it is important is it gives an estimate of how good the random forest is without using any additional data, and without using a set-aside data set.

Consider this, once you make a machine learning model, how do you know how good it is? Well, one way would be to set aside a portion, say 10%, of your training data before you train on it. After fitting your machine learning algorithm to your training data, you can run the 10% set aside through your classifier and find out if those predictions match the actual values. This is a technique known as "Cross Validation" and it is extremely useful.

However, there are a few downsides to cross-validation. Setting aside some data means that you are training on only a subset of your model. If you have a small quantity of data, setting some aside could have a large impact on the results.

The way Random Forests work inherently set aside some of the data. When each tree is generated from the data set. It uses, on average, only 63.2% of the dataset. So the remaining 36.8% is implicitly set aside.

The data that is not being used for any given tree is known as "Out of Bag" for that tree. Note that data is only "Out of Bag" for a given tree. With a large enough Random Forest, all the data will be included in some of the trees. Since any given tree has out of bag data, we can use that data to check the quality of each tree. The average error over all the trees will be our out of bag error.

For example, let's say that we have 10 data points in our Random Forest and that it is made up of 15 trees. On average each of our points will be present in approximately 2/3 of the trees, which means they will be out of bag in approximately 1/3 of the trees. That means we expect any given data point to be out of bag for 5 trees, although some will likely be out of bag for only 3 or 4 trees, and others will be out of bag for 6 or 7.

Now imagine that we have 3 different categories we are trying to classify our data points into, category A, B, C. For each of the 10 data points, we "Predict" them, i.e. try to classify them using the Random Forest, but we only include the trees where they are out of bag for that prediction. We then ask

the question, would this point have been classified correctly?

So for a couple of the 10 points

- Point 1 is actually category A. It is out of bag for 5 trees. When tested on those 5 trees, 3 trees predict category A, 1 predicts B, 1 predicts C. The most common category is A, which is correct
- Point 2 is actually category A. It is out of bag for 7 trees. The results are 3 A, 2 B, and 2 C. The most common category here is also A, even though it is not a majority. Point 2 is also correct
- Point 3 is category B. It is out of bag for 3 points. The results are 1 B, 2 C. This point is not correct.
- Point 4 is category B. It is out of bag for 4 points, the results are 2 B, 2 C. This one is difficult because there is a tie, so it depends on how the software is implemented. In python sklearn, the tie would default to the first value in the tie (because no subsequent value is greater than that value, so it wouldn't change.) In this case, the first value in the tie is category B, which happens to match the true category. So this point would be correct in this case, but a tie could also end up being not correct.

After all 10 points, we might have 7 that were classified correctly, and 3 that were classified incorrectly. So the resulting Out Of Bag score would be .7. The out of bag error would be .3

Cross-Validation

Cross-validation is a useful concept for all types of machine learning. In the out of bag error section, we saw that cross-validation isn't strictly necessary for Random Forests since the Out of Bag error is something of a baked in cross-validation.

However, oftentimes more generic cross-validation can be more useful. On a typical cross-validation, you set aside a portion of your data and use that as a metric for how well your model works. For instance, I can set aside 20% of my data, train on the other 80% and determine the quality of my model by scoring the 20% that I set aside. To score the 20% set aside, you can use a variety of metrics, such as root mean squared error, logarithmic loss, classification accuracy, or a variety of others. If you are doing some kind of data science competition, such as on Kaggle, the metric will usually be provided for you.

The most frequent cross validation I have seen is to set aside 10% of the data 10 times and train on the other 90% each of those times, averaging the results. If you do this, all the data gets set aside once, and cross-validation multiple times will give you a consistent way of identifying any improvements in your model.

So why do you set aside data when doing a cross-validation? Because if you don't, you are likely to get an artificially high result, i.e. you will think that your model is better than it actually is. The machine learning algorithm knows what the answers are in the training set, and it designs its classification to match those answers. If you test on the same data that you use to train, you will not have an accurate representation of how well the classifier would do on data it hadn't seen before. This is similar to if you are studying for an exam, and you start by looking at the answers for all of your study problems. If you then quiz yourself on those exact same questions that you already know the answer to, you will most likely do very well. But have you really learned the concept, or just memorized those specific answers? When you take the exam and have questions you've never seen before, you might discover that your self-assessments did not provide an accurate representation of your knowledge.

Below is an example of cross-validation where the classifier is trained on 90% of the model, and the other 10% is used to score it, 10 times. The code

below does this by randomizing the order of the input data (shuffling) and then slicing out the first 10%, then the next 10% etc. There are other ways to do the cross-validation, including some that are baked into python and require fewer lines of code, such as KFold which duplicates the slicing described in this paragraph. This page http://scikit-learn.org/stable/modules/cross_validation.html has a lot more information on the different cross-validation options available in python.

```
import numpy as np
from sklearn.datasets import load_iris
from sklearn.ensemble import RandomForestClassifier

# Set the random seed so that the results are always the same
np.random.seed(12345)

# Load the Iris dataset, which is already included in sklearn
iris = load_iris()
X = iris.data
y = iris.target

model = RandomForestClassifier()

# Shuffle the indices in order to get a random order for
# the cross validation
idx = np.arange(X.shape[0])
np.random.shuffle(idx)

# 10 K-fold cross validation
k_fold = 10
slice_size = X.shape[0] / float(k_fold)
```

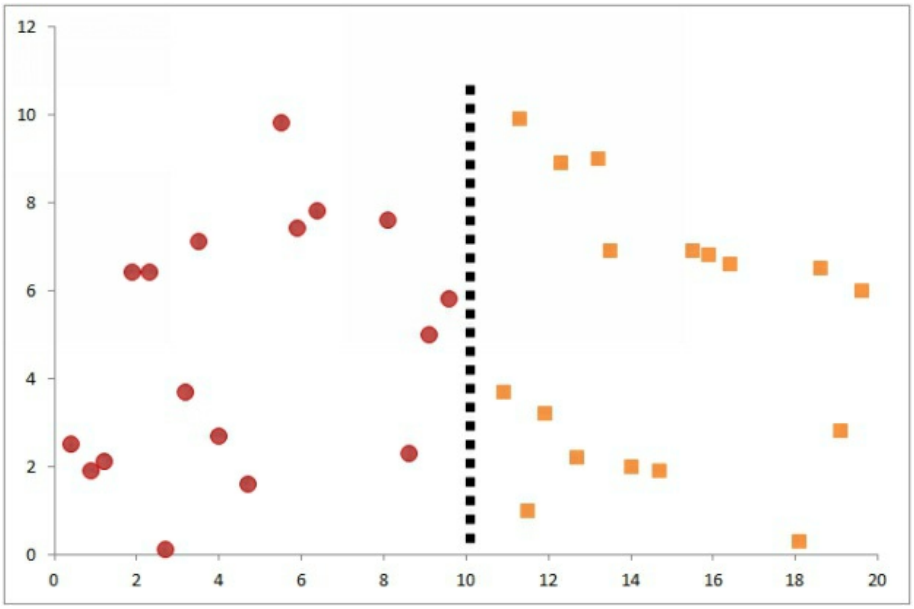
```
# loop through different cross validations
score_sum = 0.0
for i in range(0,k_fold):
    start = int(i * slice_size)
    end   = int((i+1) * slice_size)

    # slice a section from the training set
    X_train = X[ np.append(idx[0:start],idx[end:]) ]
    y_train = y[ np.append(idx[0:start],idx[end:]) ]

    # use that section for the test set
    X_test = X[ idx[start:end] ]
    y_test = y[ idx[start:end] ]

    # Train the model
    clf = model.fit(X_train, y_train)
    # score the model on the test set
    score = clf.score(X_test, y_test)
    score_sum += score

print "Average Cross Validation Score ",score_sum / k_fold
```

How A Decision Tree Picks Its Splits

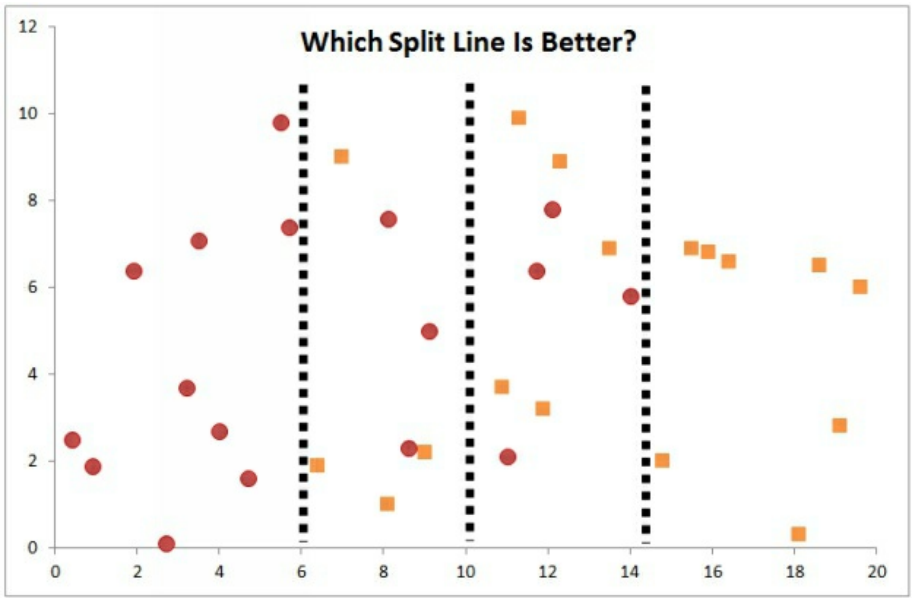
One interesting thing about each decision tree is how the threshold for each decision is generated. Despite the name "Random Forest", the thresholds are not random. They are specifically chosen to maximize the information gain at each step. That means for whatever criteria is chosen, it will split at the best location given the data in that branch.

The obvious question becomes "What is best?"

To illustrate this point, consider two examples. Say you were trying to place a vertical line to split the red circles from the orange squares.



In the first example, it is easy and obvious. Place a vertical line in the middle and you split all of the red circles from the orange squares. However, in the example below, it's harder.



If we place a line in the middle, we split most of the red circles from most of the orange squares, but is it the best possible split? Would it be better to slide farther right and pick up all the red circles on the left, but also more orange squares? Or would it be better to move farther left and get more orange squares on the right but also more red circles?

The solution comes from “Information Theory” which is a field of study in its own right.

The most commonly used solution is either the “Gini” criteria or the “Entropy” criteria. The next few pages give examples of both equations which are similar, but a little different. However, at the end of the day, it usually makes very little difference which one you use, as they tend to give results that are only a few percent different.

In Python, the default is the “Gini” criteria, so we’ll start with that.

Gini Criteria

The equation for the Gini Impurity is

$$Gini = 1 - \sum_j p_j^2$$

Where p is the probability of having a given data class in your data set.

For instance, let's say that you have a branch of the decision tree that currently has Apples, Bananas, and Coconuts in it. This leaf currently has

- 10 Apples
- 6 Bananas
- 4 Coconuts

The probability for each class is

- .5 – Apples
- .3 – Bananas
- .2 – Coconuts

So the Gini impurity is

$$Gini = 1 - (.5^2 + .3^2 + .2^2)$$

Which is equal to .62. The best value that we could have is an impurity of 0. That would occur if we had a branch that is 100% one class since the equation would become 1 – 1.

Now let's say that the decision tree has the opportunity to split this branch into two branches. One of the possible splits is

First Branch

- 10 Apples
- 5 Bananas

Second Branch

- 1 Banana
- 4 Coconuts

Just from looking at it, it seems like this split is pretty good since we are mostly separating out the apples from the coconuts, with only some bananas in each group keeping them from being a single class. But even though this split is pretty good, how do we know if it is better than

Alternative First Branch

- 10 Apples
- 6 Bananas
- 1 Coconut

Alternative Second Branch

- 3 Coconuts

The answer is to calculate the Gini Impurity for both possible splits and see which one is lower.

For the first possible split, we calculate the Gini Impurity of Branch 1 to be .444 and the Gini Impurity of Branch 2 to be .32

First Alternative - Branch 1			
Class	Count	Percentage	Square of Percentage
Apples	10	0.667	0.444
Bananas	5	0.333	0.111
Total	15	Total	0.556
		Gini Impurity - This Branch	0.444
First Alternative - Branch 2			
Class	Count	Percentage	Square of Percentage
Bananas	1	0.2	0.04
Coconuts	4	0.8	0.64
Total	5	Total	0.68
		Gini Impurity - This Branch	0.32
		Weighted Gini Impurity	0.413

There are 15 items in branch 1, and there are 5 items in branch 2. So we can calculate the combined Gini impurity of both branches by taking the weighted average of the two branches = (15 * .444 + 5 * .32) / 20 which gives a total value of .413.

The .413 impurity is a big improvement from the .62 impurity that we had before we did this split. But is it better than the other split that we could have made?

Calculating the impurity for that split,

Second Alternative - Branch 1			
Class	Count	Percentage	Square of Percentage
Apples	10	0.588	0.346
Bananas	6	0.353	0.125
Coconuts	1	0.059	0.003
Total	17	Total	0.474
		Gini Impurity - This Branch	0.526
Second Alternative - Branch 2			
Class	Count	Percentage	Square of Percentage
Coconuts	3	1	1
Total	3	Total	1
		Gini Impurity - This Branch	0
		Weighted Gini Impurity	0.447

We get a total Gini Impurity of .447, which is higher than the first alternative of .413. So with these two alternatives, a decision tree would be generated with the first choice, yielding one branch with 10 apples, 5 bananas, and a second branch with 1 banana and 4 coconuts.

Depending on your settings when generating the random forest, it likely would not stop there. It would likely continue trying to make more branches until each of them had an impurity of 0. For instance, it would take the 10 apples, 5 bananas branch, and try to find the best split for that, and continue splitting on each child branch until either all the branches had an impurity of 0, there were no more criteria to split (i.e. points had exactly equal values in all criteria), or until it hit a stopping parameter like minimum leaf size or maximum number of splits it could make.

Jumping back to the original question of “Which line is best at splitting the red circles from the orange squares?” it turns out the left most of the 3 lines would be the best based on the Gini Criteria. On the left side of the line, there were 10 red circles, 0 orange squares. On the right side of the line, there were 7 red circles, 17 orange squares. This yields a Gini Impurity of .292. The center line had an impurity of .36, and the right line had an impurity of .346, so they were both worse.

Entropy Criteria

The examples above showed how the splits would be calculated using the Gini Impurity. An alternative criteria for determining the splits is Entropy.

The equation for entropy is different than the Gini equation, but other than that, the process is pretty much the same. So if you are reading this book to get an intuitive understanding of how random forests work, and don't need to know all of the details of all the equations, this is a good section to skip and go to the next section, on feature importances.

The equation for entropy is

$$Entropy = \sum_j -p_j * log_2(p_j)$$

For entropy, just like the Gini criteria, the lower the number the better, with the best being an Entropy of zero.

For this equation, for each probability, we are multiplying that probability by the base 2 logarithm of that probability. Since each of these probabilities is a decimal between 0 and 1, the base 2 logarithm will always be negative (or zero), which when multiplied by the negative sign in the equation will give a positive number for the total entropy summation.

If we go back to the scenario where we have

- 10 Apples
- 6 Bananas
- 4 Coconuts

We can calculate the total entropy to be 1.485 as shown below

Initial Branch			
Class	Count	Percentage	-Percent * Log2(Percent)
Apples	10	0.5	0.500
Bananas	6	0.3	0.521
Coconuts	4	0.2	0.464
Total	20	Entropy Sum	1.485

Note – if you have more than 2 different classes in a branch you can get an entropy greater than 1.0. You will get the maximum entropy if every class has the same probability. In this case, if we had 33% of each of the categories we would have had an entropy of 1.585. If we had 10 categories, each with a 10% probability, the entropy would be 3.322

Looking again at the possible splits of this branch, we assume that we could split it in one of two ways (the same ways as in the Gini example) either

First Branch

- 10 Apples
- 5 Bananas

Second Branch

- 1 Banana
- 4 Coconuts

Or

Alternative First Branch

- 10 Apples
- 6 Bananas
- 1 Coconut

Alternative Second Branch

- 3 Coconuts

If we calculate the entropy for both of those possibilities, for the first possible split we get

First Alternative - Branch 1			
Class	Count	Percentage	-Percent * Log2(Percent)
Apples	10	0.667	0.390
Bananas	5	0.333	0.528
Total	15	Entropy Sum	0.918
First Alternative - Branch 2			
Class	Count	Percentage	-Percent * Log2(Percent)
Bananas	1	0.2	0.464
Coconuts	4	0.8	0.258
Total	5	Entropy Sum	0.722
		Weighted Entropy	0.869

To find the total entropy between two branches, we are using the weighted sum of the two branches. In this case, (15 * .918 + 5 * .722) / 20 = .869

For the second possible split, we get

Second Alternative - Branch 1			
Class	Count	Percentage	-Percent * Log2(Percent)
Apples	10	0.588	0.450
Bananas	6	0.353	0.530
Coconuts	1	0.059	0.240
Total	17	Entropy Sum	1.221
Second Alternative - Branch 2			
Class	Count	Percentage	-Percent * Log2(Percent)
Coconuts	3	1	0.000
Total	3	Entropy Sum	0.000
		Weighted Entropy	1.038

So for this example, just like for the Gini criteria, we see that the first possible split has a lower entropy than the second possible split, so the first possibility would be the branching that was generated.

The total information gain for the first split would be the entropy before splitting minus the entropy after splitting. Which is 1.485 - .869 = .616

Feature Importance

One of the interesting things that can be done with Random Forests is to tell you the relative importance of different features in your data. For instance, in the fruit dataset, we have three different features, length, width, and color. For this limited number of features, we can look at all of them. It is easy to plot them and would be easy to look for outliers in the data.

However many real-life data sets might have dozens or hundreds of different features. If you were trying to predict the sales price of a house, you might have features such as the size of the house, apparent quality, the location of the house, what school districts it was near, etc. In fact, even some of those features might have multiple ways that they can be expressed, for instance, the location could be street address plus city, or it could be GPS coordinates.

When you start to get a large number of features, it is useful to know which ones are the most important so you can focus your attention on them. You might have limited time but you need to scrub the outliers and errors out of your house price data set. Should you focus on the location of the house, the size of the house, or the color of the house? Intuitively you probably know that the color of the house is the least important feature of the three, but what about the other two? They both seem pretty important.

Random Forests have a couple of different ways of determining which features are the most important. And conceptually they are pretty easy to understand.

The first way to determine feature importance is the default method in Python, and it is also available in R. This method uses the information gain at each stage of the tree to determine the relative information gain between different features. The details of how information gain is calculated are described in the previous section.

For a single tree in the Random Forest, the algorithm works as follows

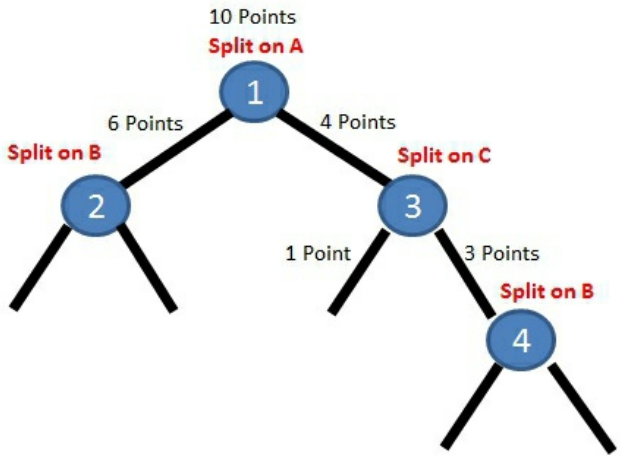
- Start with an array that is the same size as the number of features in your model. Initialize that array to zero.
- Begin to traverse the tree, with the data that was used to build the tree.
- Whenever you reach a branch in the tree, determine what feature

that branch operated on. Recall that every branch splits on one and only one feature. Additionally determine how many data points reached this branch, as opposed to following a different path down the tree

- Calculate the information gain after branching as opposed to before the branch. This is true regardless of the method used to determine information gain (e.g. Gini, Entropy, MSE, …)
- Multiply the information gain by the number of data points that reached this branch, and add that product to the array at whatever feature is being split on
- Once all the information gain for all the branches are summed, normalize the array

Once you have computed the feature importances for a single tree, you can repeat for all the trees in the Random Forest, and the average of the values to get the feature importances for the full Random Forest. This method of generating feature importances lends itself to being computed at the same time that each tree is constructed.

As an example, let's say that you have a single tree that has 10 data points, and 3 features A, B, C. Note that we have 3 features, but this example doesn't tell us the different categories of the data, it could be 2 categories or 5 categories, we don't know. The tree has 4 splits in it and looks like this



Using a made up amount of information gain at each of the splits, we can calculate the feature importance as shown in the tables below. (Since we don't actually know the categories of data, and how they were split in this

example, we can't calculate the real information gain like we did in the information gain section, so we are using made up numbers to illustrate how the feature importance works.) First, we calculate the weighted information gain at each split, weighting by the number of nodes that that branch operates on

Feature Importance

Split #	# of Data Points	Split on Which Feature	Information Gain	# of Nodes * Information Gain
1	10	A	0.26	2.6
2	6	B	0.4	2.4
3	4	C	0.3	1.2
4	3	B	0.1	0.3

And then we sum the quantities based on which feature is being used. In this example, there is only one split on the A & C features, so they are just their values as shown in the table above, and there are two splits based on feature B, so it has an importance of 2.7, which is the sum of the 2.4 and .3 values for each of the individual splits.

Importance

Feature	Importance	Normalized
A	2.60	0.40
B	2.70	0.42
C	1.20	0.18

After normalizing the values, we can see the relative importance of the features. For this example feature B was slightly more important than feature A, and they were both more important than C.

If we calculated feature importances for a different tree in our Random Forest, we might get different importances, so we would average the values on all the trees.

The Second Way To Get Feature Importance

There is not one single accepted way of generating feature importance. Different software tools use different methods. The second common way of generating feature importance makes use of the Out of Bag error feature. This method is available in the R language, but not currently available in python sklearn.

A Random Forest can generate an Out Of Bag error. That means, for all of the data points that were not selected for that tree, run those through the decision tree and determine if the decision tree classifies them correctly or not.

This method of generating feature importance makes use of the Out of Bag error feature and it works as follows

- Generate the out of bag error for the base tree, without making any changes. This is the baseline out of bag error
- For every feature in the data used to generate the tree, determine the range of numbers that those features can be. i.e. What is the maximum and minimum values in the data?
- For each feature, one at a time, randomly permute the values of that feature in all the data points between the maximum and minimum possible for that feature
- Then, after only permuting a single feature, find the out of bag error
- Calculate the difference between the new out of bag error and the baseline out of bag error. The new out of bag error will almost certainly have more error than the baseline value.
- Reset the permuted features to their original values, and begin the process again by changing a new feature.
- The features which have the greatest increase in error when they are permuted are the most important features. The features with the smallest increase in error are the least important.

An analogy for this method might be if someone was assembling a machine, or maybe a LEGO construction, you could have them do it over and over again but each time withhold a different type of part from them. At the end, you see how badly the machine turned out. If it fails catastrophically, then

the parts you withheld must have been pretty important. If the machine gets built and you can't even tell the difference, then the parts must not have been very important.

Feature Importances – General Information

Regardless of the method used to generate feature importances, there is some general information that is useful to know:

- Feature importances are almost always reported as normalized values. So what you are really seeing is how important a feature is relative to the other features.
- Typically the more features you have, the less important any single feature will be
- The more highly correlated features are to other features, the more they will split importances. For instance, if you are trying to predict a person's salary it is possible that their height could be an important feature. (i.e. maybe tall people earn more money). However if you also have features such as their shoe size and what size pants they wear in the dataset, which are both highly correlated with height, then the feature importance of height will get diluted since some of the data segmentation will be done on the other similar features.

What To Do With Named Categories, Not Numbers

Random Forests always base their splits on a single criteria and a specific numeric threshold. However, some data that you get is not numeric but is categorical. In the fruit example, we had different colors. If you were trying to categorize different animals, you might have categories for what type of skin they have such as fur, feathers, or scales. Those types of categories can be very important for classifying data, but Random Forests cannot work on them directly. They first need to be converted into numeric values.

The simplest way to convert categories into numeric values is to just assign a number for each different item in a category. For instance, animals with fur get number 1, animals with feathers get number 2, and scales get number 3. That will work well, and the Random Forest algorithm could easily split off one of the categories from the others.

Sometimes, however, you can more directly convert the categories into numbers. If you are trying to classify dogs into different breeds of dog, instead of small, medium, and large, you could input the actual weight of the dog. If you have colors, you could input the electromagnetic wavelength instead of the color, i.e. 650 nanometers instead of red, 475 nanometers instead of blue, etc. Anytime you can input this kind of more precise information, you are giving the algorithm the opportunity to get a better answer.

Limitations of a Random Forest

The Random Forest classifier and the Decision Tree classifiers that have been discussed don't do extrapolation. For example that if you have data points that show your earnings based on hours worked

- 1 Hour you earn $ 10
- 2 Hours you earn $ 20
- 3 Hours you earn $ 30

If you use the random forest classifier algorithm to try to estimate how much you will earn after 10 hours, you will not get the $100 you expect. It will probably give you $ 30 based on where it places the threshold.

Final Random Forest Tips & Summary

Here are a few final tips to make better use of Random Forests

- Use cross-validation or out of bag error to judge how good the fit is
- Better data features are the most powerful tool you have to improve your results. They trump everything else
- Use feature importances to determine where to spend your time
- Increase the number of trees until the benefit levels off
- Set the random seed before generating your Random Forest so that you get reproducible results each time
- Use a predict() or a predict_proba() where it makes sense, depending on if you want just the most likely answer, or you want to know the probability of all the answers
- Investigate limiting branching either by
    - Number of splits
    - Number of data points in a branch to split
    - Number of data points in the final resulting leaves
- Investigate looking at different numbers of features for each split to find the optimum. The default number of features in many implementations of Random Forest is the square root of the number of features. However other options, such as using a percentage of the number of features, the base 2 logarithm of the number of features, or other values can be good options.

If You Find Bugs & Omissions:

We put some effort into trying to make this book as bug-free as possible, and including what we thought was the most important information. However, if you have found some errors or significant omissions that we should address our contact information is located here

http://www.fairlynerdy.com/about/

So please send us an email and let us know! If you do, then let us know if you would like free copies of our future books. Also, a big thank you!

More Books

If you liked this book, you may be interested in checking out some of my other books such as

- Machine Learning With Boosting – Random Forests use decision trees that are all created in parallel and then averaged together. Gradient Boosted Trees also use decision trees, but they are created sequentially, with each one correcting the error in previous trees. Boosting has become one of the most dominant algorithms in data science competitions, so it is worth knowing about
- Linear Regression and Correlation – This book walks through how to do regression analysis. In doing so it demonstrates what it means when things are correlated. It also shows how to do multiple regression analysis, in a way that is completely understandable.
- Hypothesis Testing – A Visual Introduction To Statistical Significance – Hypothesis testing is how you tell if something is statistically significant, or if it likely just occurred by random chance. This book demonstrates the different types of hypothesis testing, and how you do each one.

And if you want to get any of the python programs shown in the book, the python programs that generated the plots, or the Excel files shown in this book, they can be downloaded at http://www.fairlynerdy.com/random-forest-examples/ for free

Thank You

Before you go, I'd like to say thank you for purchasing my eBook. I know you have a lot of options online to learn this kind of information. So a big thank you for reading all the way to the end.

If you like this book, then I need your help. **Please take a moment to leave a review on Amazon.** It really does make a difference, and will help me continue to write quality eBooks on Math, Statistics, and Computer Science.

P.S.

I would love to hear from you. It is easy for you to connect with us on Facebook here

https://www.facebook.com/FairlyNerdy

or on our webpage here

http://www.FairlyNerdy.com

But it's often better to have one-on-one conversations. So I encourage you to reach out over email with any questions you have or just to say hi!

Simply write here:

ImFairlyNerdy (at) gmail (dot) com

~ Scott Hartshorn