Trees, Bagging, and Random Forests

Gordon Ross

Decision Trees

Decision trees are an alternative approach to both classification that can fit non-linear models

The basic idea of trees is to segment the feature space into a number of smaller, simpler regions. Each small region is then assigned to a single class (in a classification problem).

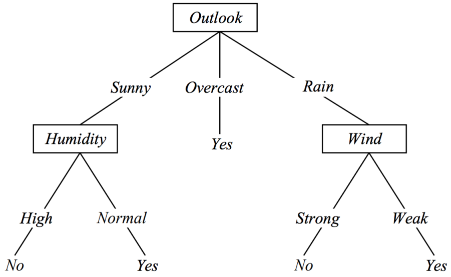
Decision Trees

Suppose we have a binary classification problem where we are trying to learn whether two people will play/not play tennis on a particular day, depending on the weather conditions. We have access to 14 previous days of data:

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Decision Trees

A decision tree for predicting whether tennis will be played (Yes/No) looks like this:



Decision Trees

The bottom parts of the tree are called leaf nodes, and correspond to the classification we are making (i.e. yes or no).

Once a tree like this has been learned, we can predict future observations (eg the test set) by just going down the tree.

For example, if the Outlook ion a new day is overcast, our prediction will be to play tennis.

If the Outlook is raining, then we go down that branch of the tree, and then predict tennis is played if the Wind is weak, and not played if it is strong.

Decision Trees

Now lets discuss how we actually build trees from a given training set

There are several different algorithms for building decision trees. We will look at the **ID3** algorithm. This is used for classification and is only applicable when the features are discrete (not continuous).

The ID3 algorithm constructs the tree greedily, starting with the root node at the top of the tree. It then constructs the tree layer by layer, each time picking a single one of the features to split on

Decision Trees

So, lets build a tree for the PlayTennis data. Recall it looks like this:

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

The first step is to choose which feature to put at the top of the tree (called the root node).

Decision Trees

There are several different criteria to choose which feature to use. In ID3, we use a criteria based on **entropy**.

Definition: Suppose a discrete random variable X takes on R possible values $v_1, v_2, \ldots, v_R$. Suppose that $p(X = v_r) = p_r$.

Then, the entropy of X is defined as $-\sum_r p_r \log_2 p_r$ (note this is log to the base of 2).

Decision Trees

Intuitively, the higher the entropy of a random variable, the more unpredictable it is.

For example, suppose X can only take on two values, 0 and 1. If $p(X = 0) = 0.9$ then the entropy is:

$$-0.9 log2(0.9) - 0.1 log2(0.1) = 0.47$$

Now suppose instead that $p(X = 0) = 0.5$. The entropy is:

$$-0.5 log2(0.5) - 0.5 log2(0.5) = 1$$

In this case, the outcome is harder to predict

Decision Trees

In the ID3 algorithm, we choose which variable to split on, in order to maximise the information that we gain at that stage.

Formally, the information gain of a split is defined as the increase in entropy that comes from making the split.

Information gain is defined as:

$$\mathit{Entropy}(\mathit{parent}) - \mathit{AverageEntropy}(\mathit{children})$$

This is easier to show by example.

Decision Trees

The entropy of this data (looking at the PlayTennis variable that we are trying to predict) is:

$$-9/14 log2(9/14) - 5/14 log2(5/14) = 0.94$$

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Decision Trees

Suppose we decided to split on the Wind variable. This variable has 2 possible values: Strong and Weak. We look at the conditional distribution of PlayTennis in both of these cases:

- When Wind=Weak, we have 6 Yes outcomes and 2 No outcomes. So the entropy is: $-6/8log2(6/8) - 2/8log2(2/8) = 0.81$
- When Wind=Strong, we have 3 Yes outcomes and 3 No outcomes. So the entropy is: $-3/6log2(3/6) - 3/6log2(3/6) = 1$

The weighted entropy is $8/14 * 0.81 + 6/14 * 1 = 0.89$ (since the Weak outcome occurs 8/14 times, and Strong occurs 6/14 times).

So the information gain from splitting on Wind is equal to $0.94 - 0.89 = 0.05$

Decision Trees

Suppose we decided to split on the Humidity variable. This variable has 2 possible values: High and Normal. We look at the conditional distribution of PlayTennis in both of these cases:

- When Humidity=High, we have 3 Yes outcomes and 4 No outcomes. So the entropy is: $-3/7 log2(3/7) - 4/7 log2(4/7) = 0.985$
- When Humidity=Normal, we have 6 Yes outcomes and 1 No outcomes. So the entropy is: $-6/7 log2(6/7) - 1/7 log2(1/7) = 0.591$

The weighted entropy is $7/14 * 0.985 + 7/14 * 0.591 = 0.788$

So the information gain from splitting on Humidity is equal to $0.94 - 0.788 = 0.152$

Decision Trees

Suppose we decided to split on the Outlook variable. This variable has 3 possible values: Sunny, Overcast, Rain. We look at the conditional distribution of PlayTennis in each of the three cases:

- When Outlook=Rain, we have 3 Yes outcomes, and 2 No outcomes. The entropy of this outcome is $-3/5 log2(3/5) - 2/5 log2(2/5) = 0.97$
- When Outlook=Sunny, we have 2 Yes outcomes, and 3 No outcomes. The entropy of this outcome is $-2/5 log2(2/5) - 3/5 log2(3/5) = 0.97$
- When Outlook=Overcast, we have 4 Yes outcomes, and 0 No outcomes. The entropy of this outcome is $-4/4 log2(4/4) - 0/4 log2(0/4) = 0$ (by convention we define the entropy as 0 when all observations are in the same class, since $log2(0) = -\infty$

The weighted entropy is $5/14 * 0.97 + 5/14 * 0.97 + 4/14 * 0 = 0.69$ (since the Rain outcome occurs 5 times out of 14, and so on).

So the information gain is $0.94 - 0.69 = 0.25$

Decision Trees

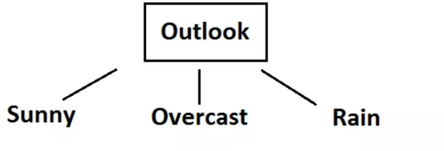
If we look at all the attributes, the information gains for the splits are:

- Outlook: 0.25
- Temperature: 0.029
- Humidity: 0.15
- Wind: 0.05

So. we should split on the Outlook variable. This gives us our first split, at the top of the tree.

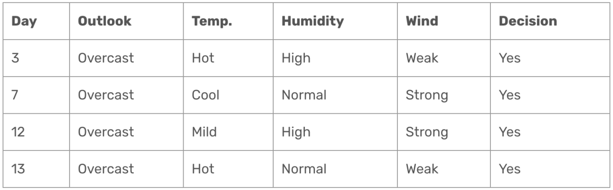
Decision Trees

So the tree looks like this, so far:



Decision Trees

Now we repeat to find the next splits. First consider the case where Outlook. = Overcast. The data looks like this (looking only at the rows where Outlook = Overcast)

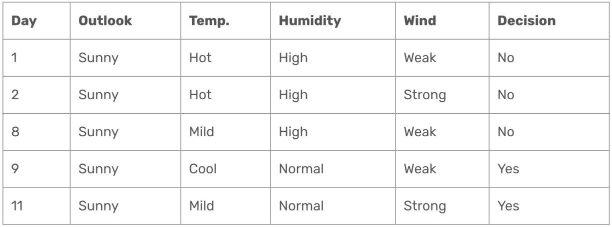


Day	Outlook	Temp.	Humidity	Wind	Decision
3	Overcast	Hot	High	Weak	Yes
7	Overcast	Cool	Normal	Strong	Yes
12	Overcast	Mild	High	Strong	Yes
13	Overcast	Hot	Normal	Weak	Yes

In this case we will always classify as being Yes

Decision Trees

Now look at the case where Outlook=Sunny:



Day	Outlook	Temp.	Humidity	Wind	Decision
1	Sunny	Hot	High	Weak	No
2	Sunny	Hot	High	Strong	No
8	Sunny	Mild	High	Weak	No
9	Sunny	Cool	Normal	Weak	Yes
11	Sunny	Mild	Normal	Strong	Yes

The entropy is: $-2/5log2(2/5) - 3/5log2(3/5) = 0.97$

Decision Trees

If we now split on Humidity, then the information gain is:

$$0.97 - (3/5 * 0 + 2/5 * 0) = 0.97$$

If we now split on temperature, then the information gain is:

$$0.97 - (2/5 * 0 + 2/5 * (-1/2 log2(1/2) - 1/2 log2(1/2)) + 1/5 * 0) = 0.57$$

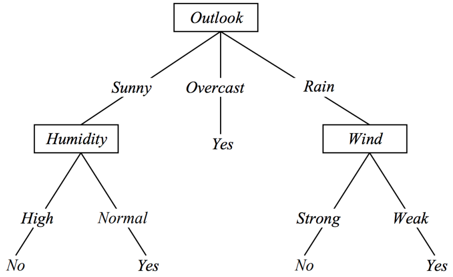
If we now split on wind, then the information gain is:

$$0.97 - (3/5 * (-2/3 * log2(2/3) - 1/3 * log2(1/3)) -$$
$$2/5 * (-1/2 * log2(1/2) - 1/2 * log2(1/2))) = 0.82$$

So we would split on Humidity

Decision Trees

Repeating the process gives the final decision tree:



We continue splitting until we reach the leaf nodes where all the observations are assigned to the same class.

Decision Trees

There are several other decision tree algorithms that can be used for classification other than ID3. These typically vary based on what criteria is used to choose which feature to split on at each stage, and in terms of what post-processing is done to the tree after it is formed.

A popular algorithm is C4.5 which is a generalisation of ID3 which lets it handle continuous as well as discrete features. It also prunes the tree (which we will discuss later) to reduce overfitting

Decision Trees

A potential problem with trees is that they overfit, i.e they learn a model which is too complicated and wont generalise well to test set data.

As such it is common to prune the tree after it has been built. This results in a smaller tree, i.e. one with less split points. This is sort of like dimensionality reduction

There are several approaches for doing this, but they usually involve discarding split points using cross-validation to assess whether this improves the cross-validated error rate.

Pros and Cons of Trees

Advantages:

- Trees allow for flexible non-linear models without the need to worry about basis functions/kernels/etc
- They are very easy to explain to non-statisticians, which is important in real-world/business applications.
- It is easy to understand which features are driving the classification process. Plotting the trees to give a visual representations makes this even clearer.

Pros and Cons

Disadvantages:

- They dont work very well.

Ok thats an extreme statement, but in practice the performance of decision trees tends to be worse than other classification algorithms.

However, combining decision trees with the **bagging** algorithm leads to the **random forests**, which is competitive with alternative approaches

Random Forests

High Variance

The main reason why decision trees sometimes have bad performance is that they are a very high variance model.

Suppose our training set consists of *n* observations, and we fit a decision tree model. If we had collected a different *n* observations (from the same distribution) then we would usually end up learning a very different tree.

Or in other words, suppose we split our training set into two equally sized parts, and fit a seperate tree to each part. It is likely that the two trees will be completely different (in the sense of making different splits and different levels of the tree).

Trees are very unstable, in that small changes to the data can produce very different trees. This is not generally the case with the other classification algorithms we have considered such as KNN.

Bagging

Suppose in general that we have a set of independent random variables $Z_1, \ldots, Z_n$ from some distribution with variance σ^2.

Then we know in general that the sample mean:

$$\bar{Z} = \frac{1}{n} \sum_i Z_i$$

has variance σ^2/n.

In other words, averaging a set of observations reduces variance.

Bagging

We can apply this insight to classification as follows: rather than fitting one classifier to the data, suppose we instead collect several different training sets, and fit a separate classifier (eg a separate tree) to each one. Say we fit M classifiers this way.

Given a new observation x (e.g. in the test set), let $f^j(x)$ be the prediction of the j^{th} classifier where $j \in 1, \ldots, M$.

Bagging

In a classification.problem, we would predict x by classifying the point using all M trees separately, and then assigning to the class which got the most ‘votes’ (i.e the one which most of the trees predicted).

This is known as a majority vote algorithm. It has lower variance than using a single tree.

Bagging

But in practice we only have one training set. Collecting extra data is typically expensive, so we cant just use extra training sets.

What if we instead split our training set into different subsets? Eg if we have 100 training observations then we could split them into 10 different subsets each containing 10 observations, and fit a different classifier to each one.

In general this is a bad idea since the resulting training sets will contain too few observations to allow an effective classifier to be learned.

Bagging

Instead we can use **boostrapping**. This is a very general technique in statistics which creates additional replica datasets using resampling.

Basic idea: given a set of n observations, we create a new data-set by randomly selecting n of these observations **with replacement**. So, each of the original observations can occur in the new data set more than once.

This new data-set is called a bootstrapped data set. We can then repeat the process and generate a new dataset by again sampling n observations from the original data with replacement.

By repeating this over and over, we can get M replica data sets from the original trianing data

Bootstrapping

Example: suppose we have 5 observations 2.4, 3.2, 2.2, 4.6, 3.3.

We draw M bootstrap data sets each of size $n = 5$ by assigning each observation an equal probability of being selected (1/5), and repeatedly drawing samples of M observations. So we might get (e.g.):

- Bootstrap-sample 1: 3.3, 3.2, 2.4, 4.6, 2.2
- Bootstrap-sample 2: 2.2, 2.2, 2.4, 3.2, 2.4
- ...
- Bootstrap-sample M: 3.2, 4.6, 2.4, 4.6, 4.6

Sampling is done **with replacement** so each observation can be selected multiple times in each pseudo-sample

Choose M to be large, e.g $M = 1000$ (exact value does not matter)

Bagging

This leads to the general bootstrap aggregation algorithm, also known as **bagging**. Note that this can be applied to any classifier, not just decision trees

- Create M bootstrap training sets by applying the.bootstrap to the original training set
- For each bootstrap training set j, train a classifier using only this data.
- When presented with a new observation x, let $f^j(x)$ by the prediction when using the j^{th} classifier.
- Our final prediction for x is the majority-vote class

Bagging

The key reason why bagging works is that even though each individual classifier that we learn will be worse than if we had trained a single classifier on all the data, their overall average performance should be better due to variance reduction.

Although bagging can be used with any classifier, it is particularly useful for decision trees. Since the bootstrapped datasets are all generated from the same training set, they will be quite similar (even though they are each different). However the fact that decision trees are a high variance classifier means that the trees learned on each one can be quite different.

This leads to more diversity among classifiers, and hence lower variance.

Bagging

One disadvantage of bagging is that it can be computationally expensive for large data sets. However this computational burden can be partly reduced by noting that there is a simple way to estimate the test set error which doesnt involve cross-validation.

As such, even though bagging takes a longer time to run on a computer, we will often not need to do cross-validation, which can save time

The key is that (from general bootstrap theory) it is known that each bootstrapped data set contains on average around 2/3 of the total observations (with the other 1/3 being repeated observations).

Bagging

As such, on average 1/3 of observations will not be contained in each bootstrapped data-set. As such, these can be used to evaluate the performance of the tree that is fitted on that bootstrapped data-set, since they will not have been involved in training.

So for each observation i in the training set, we can predict its values using only the trees for which it was not in the training set. So on average, we will have M/3 predictions for each observation.

Averaging these together gives a final prediction for each observation, and then averaging over the whole data set gives an estimate of the test set (i.e out-of-sample) error

Random Forests

Bagging tends to improve the performance of trees substantially. However, performance can be improved even further. In order to maximally decrease the overall variance, we ideally want the M trees that we learn to be as different to each other as possible.

In standard bagging, the trees will be somewhat similar since there is so much overlap among the different bootstrapped datasets (since they were all generated from the same original training set).

We can use the following tweak to the bagging algorithm. Suppose that there are K features in the classification problem. For each bootstrapped data-set, rather than using all K of these features, instead choose a random subset $P < K$ features to use.

Each bootstrapped dataset now uses a different set of features. This ensures the trees will be very different

Random Forests

The resulting algorithm is known as **random forests**. Suppose we have n observations and K features

- Create M bootstrap training sets by applying bootstrap to the original training set.
- For each bootstrap training set, choose P features at random and discard all the others for that set.
- For each such bootstrap training set j, train a classifier using only this data.
- When presented with a new observation x, let $f^j(x)$ by the prediction when using the j^{th} classifier.
- Our final prediction for x is the majority-vote class.

Conventional wisdom says that for classification problems, we should choose $P = \sqrt{K}$

Ensemble Learning

Bagging is an example of **ensemble learning**: rather than learning a single classifier, we instead learn multiple classifiers and average their predictions.

The intuition behind ensemble methods is that if we learn multiple classifiers, then each one will be good at predicting a different subset of the data.

Random forests are probably the most well-known and widely used example of an ensemble classifier

An alternative (and less widely used) approach to bagging is called **boosting**, particularly the Adaboost algorithm. However this tends not to be used much these days, so we will not cover it here

Stylometry

Let's apply random forests to some of the stlyometry problems that we have looked at. We will use the randomForest() function in the **randomForest** package:

```
install.packages('randomForest')
library(randomForest)
```

Stylometry

When using discriminant analysis and KNN, we combined all the books that each author wrote together, i.e. each author was represented as a single vector which counted their total function words across all their books.

When applying random forests, we will treat the data differently: each individual book will be an observation. So we have a separate observation (i.e. function word vector) for each book written by each author

Stylometry

We first load the corpus. After that, we create a matrix **x** where each row is the function word count of a single book. The vector **y** is the class label for the author of each book.

```
M <- loadCorpus("~/Dropbox/Teaching/SCS/Data/Corpus
  /FunctionWords/", "frequentwords70")

x <- NULL
y <- NULL
for (i in 1:length(M$features)) {
  x <- rbind(x,M$features[[i]])
  y <- c(y,rep(i,nrow(M$features[[i]])))
}
```

Stylometry

Next we standardise the data by converting counts to proportions, and rescaling to have mean 0 and standard deviation 1:

```
for (i in 1:nrow(x)) {
  x[i,] <- x[i,]/sum(x[i,])
}

for (j in 1:ncol(x)) {
  x[,j] <- (x[,j] - mean(x[,j])) / sd(x[,j])
}

y <- as.factor(y)
```

Stylometry

We can do leave-one-out cross validation as follows (note this will take about a minute to run):

```
preds <- numeric(nrow(x))
for (i in 1:nrow(x)) {
  print(i) #there are 168 books in this corpus
  rf <- randomForest(x[-i,],y[-i])
  preds[i] <- predict(rf,x[i,])
}
sum(preds==y)/length(y)
[1] 0.922619 #accuracy, around 92%
```

Stylometry

As with discriminant analysis and KNN, I have provided a randomForestCorpus() function which implements the above code, and which you can use instead.

It has the same format as the discriminantCorpus and KNNCorpus functions you have already seen: simply pass it in the training and test sets as lists, and it will return the predictions. You therefore don't need to do the above standardisation/etc manually.

Let's illustrate this using the Federalist Papers corpus (you can use the same code for the above corpus also).

Stylometry

```
M <- loadCorpus("~/Dropbox/Teaching/SCS/Data/
  Federalist/FunctionWords/", "frequentwords70")
M$authornames

train <- M$features[-5]
test <- M$features[5] #unknown texts
test <- test[[1]] #needed due to how lists are constructed

> discriminantCorpus(train,test)
 [1] 4 4 4 4 4 4 1 4 4 4 4 4
> KNNCorpus(train,test)
 [1] 4 4 4 4 4 4 1 4 4 4 4 4
> randomForestCorpus(train,test)
 [1] 4 4 4 4 4 4 1 4 4 4 4 4
```

So random forests agrees with the classifiers we have already seen.

Stylometry

We can implement leave-one-out cross validation to assess how well random forests performs on this corpus, looking only at the texts with known authorship. This has the same format as the discriminant analysis and KNN code you have already seen:

```
RFpredictions <- NULL
truth <- NULL
features <- M$features[-5] #discard unknown texts
for (i in 1:length(features)) {
  for (j in 1:nrow(features[[i]])) {
    testdata <- matrix(features[[i]][j,],nrow=1)
    traindata <- features
    traindata[[i]] <- traindata[[i]][-j,,drop=FALSE]

    pred <- randomForestCorpus(traindata, testdata)
    RFpredictions <- c(RFpredictions, pred)
    truth <- c(truth, i)
  }
}
```

Stylometry

```
sum(RFpredictions==truth)/length(truth)
[1] 0.9178082
```

So random forests achieves 91.7% accuracy on this corpus

Hyperparameters

Remember that with KNN, we had some freedom to select how many nearest neighbours to consider. For example, we could have K=1 or K=3.

This is a general property of machine learning algorithms – they often have some parameters which can take on different values. We have to choose particular values, and performance will depend on the parameters we select. These are sometimes known as ‘hyperparameters’.

For random forests, the hyperparameters are the number of trees to grow, and the proportion of features to incldue in each of the tree.

In the above analysis we used the default values in the R package. But these are not necessarily the best choices!

Hyperparameters

To investigate the hyperparameters, we can consult the manual page for the randomForest function:

```
> ?randomForest
```

Reading through the page, we find arguments:

ntree: Number of trees to grow. This should not be set to too small a number, to ensure that every input row gets predicted at least a few times.

mtry: Number of variables randomly sampled as candidates at each split. Note that the default values are different for classification (sqrt(p) where p is number of variables in x) and regression (p/3)

Hyperparameters

Choosing the best value of the hyperparameters of any machine learning method can be difficult!

One common approach is cross-validation. Essentially: try many different values, and evaluate the cross-validation accuracy of each choice using the training set. Then, select the hyperparameters which give the highest accuracy

We will not explore this further in this course, but its important if you ever go on to use machine learning methods in practice!