MATH5743M: Statistical Learning

Dr Seppo Virtanen, School of Mathematics, University of Leeds

Semester 2: 2022

Week 7: Decision Trees

Imagine I told you the weight and age of a number of dogs, and asked you to decide if they were either German Shepherds (a rather large breed, see Figure 1), or Jack Russel Terriers (a rather small breed, also see Figure 1). Based on just the weight and age, how might you decide?

Figure 1: A German Shepherd and a Jack Russel Terrier. The typical weight of a German Shepherd is about 25kg, the typical weight of a Jack Russel is about 5kg

Well, if you knew anything about the typical weights of such animals, you might start by assuming that all dogs weighing more 15kg are German Shepherds, and those weighing less are Jack Russel Terriers. This would give you a reasonably accurate way to predict the breed. But now I remind you that some of the dogs in the list are very young – some of those small dogs might actually be German Shepherd puppies. So now you make a more complicated decision: if a dog is more than 15kg, it must be a German Shepherd, but if it is less than 15kg you look at its age; if the age is less than 1 year and the dog is already over 5kg then it is probably a German Shepherd puppy, otherwise it is a Jack Russel.

What you have done is create an ordered sequence of decision points, based on the data you are given. This can be represented by a structure called a *decision tree*. The decision tree for the problem above is shown in Figure 2. To make a decision, one starts at the top of the tree, and uses the decision rules provided to choose a path down through the branches until reaching the end at a *leaf*. The value of this leaf then tells you what decision to make.

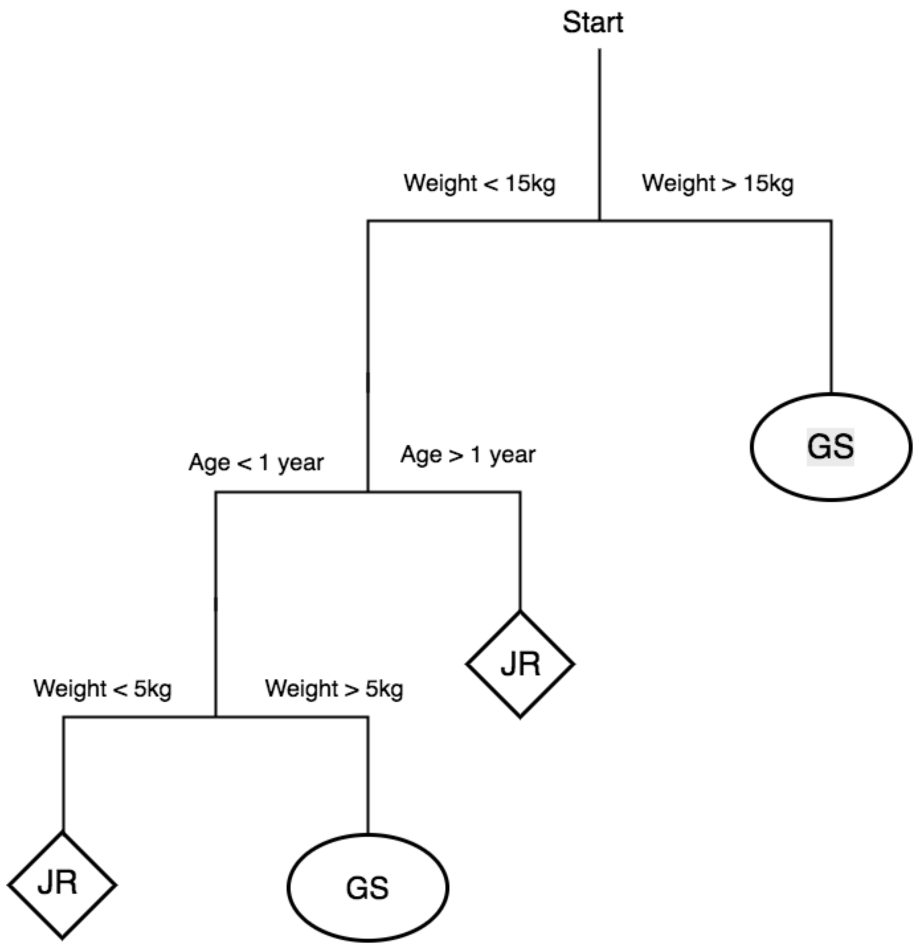


Figure 2: A decision tree representing the decision-making process for choosing the breed of a dog from weight and age.

In this example I decided on what decision rules to use in the tree by intuition, but in general when using decision trees we use an algorithmic approach to build a tree based on a training data set.

Training a decision tree

There are several possible algorithms for creating decision trees from data. We are going to focus on the most commonly used method: a tree is grown by recursively creating one split after another, based on the values of input variables, so as to maximise the probability that we classify the output variables correctly. This is best understood by demonstration. Lets return to the example above with the dogs, but now imagine that we are given some training data from which to build a tree. For the moment we will just consider the weight of the dog, and the breed. Lets imagine this data is as below

```r
print(dog_data)
```

```
##    weight   age breed
## 1       4  4.00    JR
## 2       3  5.00    JR
## 3       5  0.25    GS
## 4       8  0.50    GS
## 5       8  6.00    JR
## 6      15  7.00    GS
## 7      20 10.00    GS
## 8      25  8.00    GS
## 9       5  7.00    JR
## 10     30  6.00    GS
## 11     12 10.00    JR
## 12     22  9.00    GS
```

Now, we want to create a single split that best divides the data into two distinct species. To calculate how good a split is, we introduce a quantity called the 'Gini Impurity'. For each split, imagine that you have n_A items of class A, and n_B of class B (with corresponding proportions $\rho_A = n_A/(n_A + n_B)$ and $\rho_B = 1 - \rho_A$. What is the probability that if you pick an item at random, and classify it as either A or B with probabilities corresponding to the proportions of each item, that you will make a mistake? This is the Gini Impurity of a set, and it measures how close the set is to being composed of just one class. We can calculate it like this: to make a mistake we must choose an item of class A (with probability ρ_A) and then misclassify it with probability ρ_B, or vice versa. Both of these give the same probability, which is the Gini Impurity G:

$$G = \rho_A \rho_B = \rho_A(1 - \rho_A) \tag{1}$$

If the set is 'pure', i.e. all of one class, then $G = 0$. Alternatively if the set is half A and half B then $G = 0.25$, which is the maximum possible value. We will try splitting the data according to the weight of the dog. To identify the best first split to make in our growing tree we need to find the value of 'weight' where, if we split the data there, we minimise the average Gini Impurity of the resulting two leaves. Lets look at a few possible splits and see what the resulting Gini Indices are. First, lets consider what the Gini Impurity of the entire data set is. We find the proportion of Jack Russels in the set and calculate G from this:

```r
p = mean(dog_data$breed == 'JR')
G = p*(1-p)
print(G)
```

```
## [1] 0.2430556
```

This gives a baseline to compare possible splits to – note that it is almost as high as the maximum possible (0.25).

Now consider if we split the data at weight=16

```r
idx = which(dog_data$weight < 16)
group1 = dog_data[idx, ]
group2 = dog_data[-idx, ]
print(group1)
```

```
##    weight   age breed
## 1       4  4.00    JR
## 2       3  5.00    JR
## 3       5  0.25    GS
## 4       8  0.50    GS
## 5       8  6.00    JR
## 6      15  7.00    GS
## 9       5  7.00    JR
## 11     12 10.00    JR
```

```
print(group2)
```

```
##    weight age breed
## 7      20  10    GS
## 8      25   8    GS
## 10     30   6    GS
## 12     22   9    GS
```

Lets find the proportion of JR in group 1 and group 2

```
p1 = mean(group1$breed == 'JR')
p2 = mean(group2$breed == 'JR')
```

Now calculate the Gini Impurity of each

```
G1 = p1*(1-p1)
G2 = p2*(1-p2)
```

Finally record the (weighted) average Gini Impurity of the two groups. The weighting is important: we need to average the two impurities based on how big each new subset is:

```
G = (G1*dim(group1)[1]+G2*dim(group2)[1])/(dim(group1)[1]+dim(group2)[1])
print(G)
```

```
## [1] 0.15625
```

This is lower than for the unsplit data, but can we do better? What about we choose to split at weight=14? We can repeat the same procedure, with the new split value

```
idx = which(dog_data$weight < 14)
group1 = dog_data[idx, ]
group2 = dog_data[-idx, ]
print(group1)
```

```
##    weight   age breed
## 1       4  4.00    JR
## 2       3  5.00    JR
## 3       5  0.25    GS
## 4       8  0.50    GS
## 5       8  6.00    JR
## 9       5  7.00    JR
## 11     12 10.00    JR
```

```
print(group2)
```

```
##    weight age breed
## 6      15   7    GS
## 7      20  10    GS
## 8      25   8    GS
## 10     30   6    GS
## 12     22   9    GS
```

```r
p1 = mean(group1$breed == 'JR')
p2 = mean(group2$breed == 'JR')
G1 = p1*(1-p1)
G2 = p2*(1-p2)
G = (G1*dim(group1)[1]+G2*dim(group2)[1])/(dim(group1)[1]+dim(group2)[1])
print(G)
```

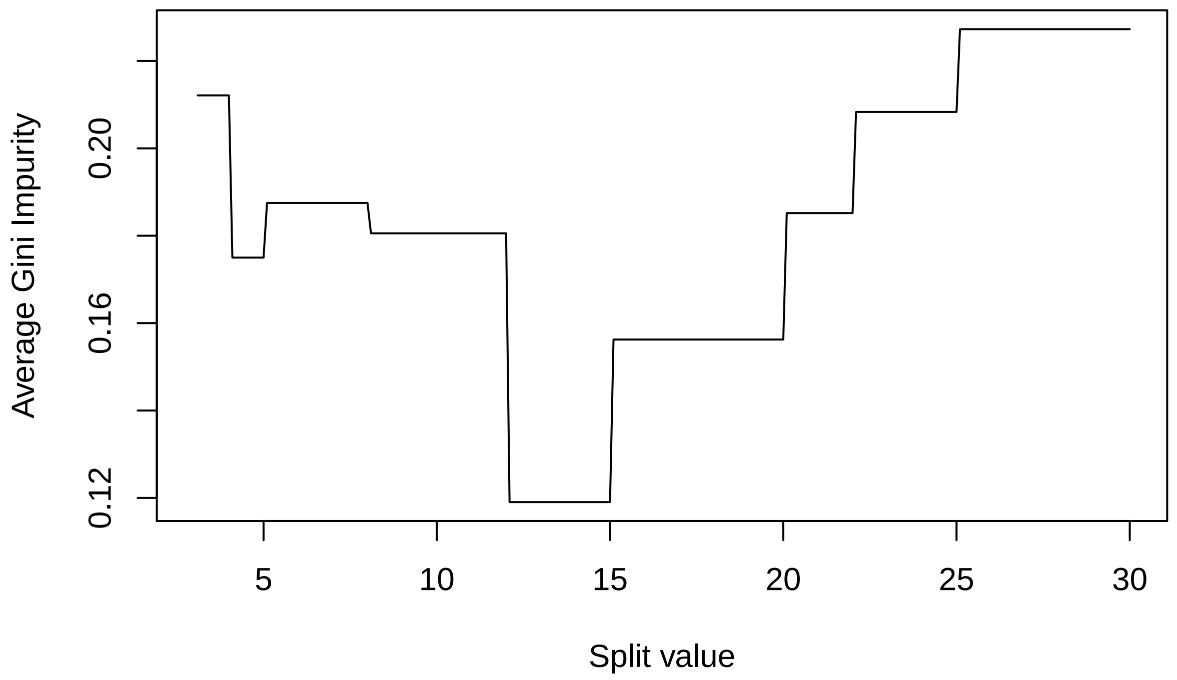
```
## [1] 0.1190476
```

Choosing to split at weight=14 rather than weight = 16 gives a much lower average Gini Impurity, and is therefore a better choice for building our decision tree. Of course, if we want to do this automatically we need to consider all possible split values. To do this we need to create a function that returns the average Gini Impurity. The function needs to know which input to use when splitting, and we will give it a default of 'weight':

```r
average_G <- function(split_value, data, input='weight'){
  idx = which(data[, input] < split_value)
  group1 = data[idx, ]
  group2 = data[-idx, ]
  p1 = mean(group1$breed == 'JR')
  p2 = mean(group2$breed == 'JR')
  G1 = p1*(1-p1)
  G2 = p2*(1-p2)
  G = (G1*dim(group1)[1]+G2*dim(group2)[1])/(dim(group1)[1]+dim(group2)[1])
  return(G)
}
```

Now we can map out the mean value of G as a function of the split value by testing many different values

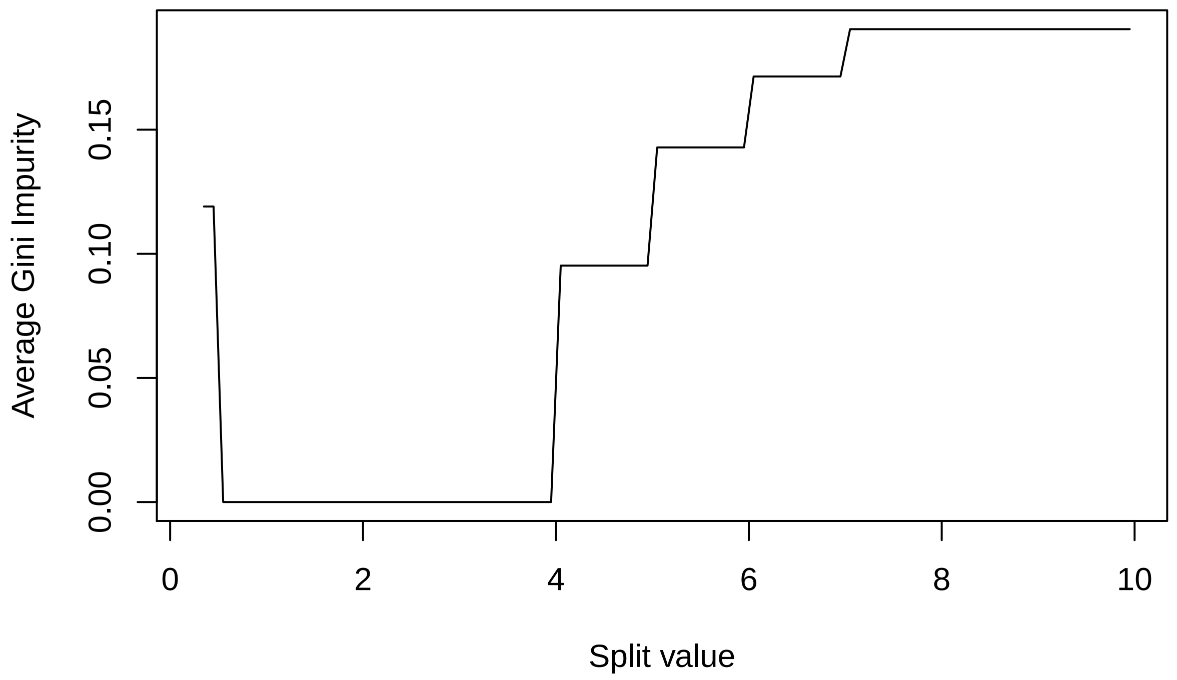
```r
s = seq(min(dog_data$weight), max(dog_data$weight), 0.1) #Possible split values
G = rep(NA, length(s)) #Initialise vector of G values
for (i in 1:length(s)){
  G[i] = average_G(s[i], dog_data) #Get the value of G for this split
}
plot(s, G, type='l',xlab='Split value', ylab='Average Gini Impurity')
```



We see that the average Gini Impurity follows a step structure - in this data a split at 20.1 or 20.9 gives the same result since all weights are integer values. If we had more densely packed values of weight then the curve would look more continuous. In this example we only considered a single input variable, and a single split at weight=13, 14 or 15 gives a very good classification – one group contained only German Shepherds, the other only Jack Russels and two German Shepherd puppies.

Having made this first split (lets say at weight = 14 as the midpoint of the range), we can now consider partitioning the group that remains mixed – the one containing both Jack Russels and German Shepherds. We repeat the procedure above on this subset, but rather than using weight, we will now try splitting the group according to age – remember this means we need to tell the function we defined above (average_G) to use age as the input, and only apply this to dogs below 15kg (group1).

```
idx = which(dog_data$weight < 14)
group1 = dog_data[idx, ]
group2 = dog_data[-idx, ]
s = seq(min(group1$age), max(group1$age), 0.1) #Possible split values
G = rep(NA, length(s)) #Initialise vector of G values
for (i in 1:length(s)){
  G[i] = average_G(s[i], group1, input='age') #Get the value of G for this split
}
plot(s, G, type='l',xlab='Split value', ylab='Average Gini Impurity')
```



We can see from these results that any split between age = 0.5 and age = 4 gives an average Gini impurity of zero, i.e. a perfect split. We can choose any value in this range, so let's choose 2.25 years as the mid point. This means that our tree is now complete. First we ask whether the dog is more than 14kg: if so, we class it as a German Shepherd; if not we look at the dog's age. Those below 14kg and older than 2.25 years are Jack Russels, while those below 14kg but younger than 2.25 years are German Shepherds. Note that this is intentionally very similar to the decision tree I proposed at the very start before we saw any data.

Now lets look at another simple toy problem in its entirety, so we can see the process from start to finish. Imagine we start with the following data:

x	1	2	3	4	5
y	0	1	0	1	1

Lets consider the places we could split the data (assuming we do so between integer values of x) and the resulting Gini Impurity:

- Split at $x = 0.5$: $G = (3/5)(2/5) = 0.24$
- Split at $x = 1.5$: $G = 0 + (4/5)[(3/4)(1/4)] = 0.15$
- Split at $x = 2.5$: $G = (2/5)[(1/2)(1/2)] + (3/5)[(1/3)(2/3)] = 0.23$
- Split at $x = 3.5$: $G = 0 + (3/5)[(2/3)(1/3)] = 0.13$
- Split at $x = 4.5$: $G = 0 + (4/5)[(1/2)(1/2)] = 0.20$
- Split at $x = 5.5$: $G = (3/5)(2/5) = 0.24$

So we should make our first split at $x = 3.5$. That gives us two new subsets:

x	1	2	3
y	0	1	0

and

x	4	5
y	1	1

in the second of these subsets all the y values are the same, so we don't need to consider splitting it any further: we can say that for $x > 3.5$, $y = 1$. Now we consider splits in the first subset of the data in the same way, going through each possible split again:

- Split at $x = 0.5$: $G = (2/3)(1/3) = 0.22$
- Split at $x = 1.5$: $G = 0 + (2/3)[(1/2)(1/2)] = 0.17$
- Split at $x = 2.5$: $G = 0 + (2/3)[(1/2)(1/2)] = 0.17$
- Split at $x = 3.5$: $G = (2/3)(1/3) = 0.22$

Here the possible splits at $x = 1.5$ and $x = 2.5$ are equally good, so we have to pick one at random. Lets choose $x = 1.5$. That splits this subset into two further subsets:

x	1
y	0

and

x	2	3
y	1	0

The first subset only has one value of y and cannot be split any further. The split in the second is now clear, but lets work it out for completeness:

- Split at $x = 1.5$: $G = (1/2)(1/2) = 0.25$
- Split at $x = 2.5$: $G = 0$
- Split at $x = 3.5$: $G = (1/2)(1/2) = 0.25$

so we split at $x = 2.5$. This leaves us with no further splits to perform - every subset we have made now contains only one value of y.

We can summarise these successive splits using a tree diagram (Figure 3). Check that following each possible branch performs the same splits as we discovered above.

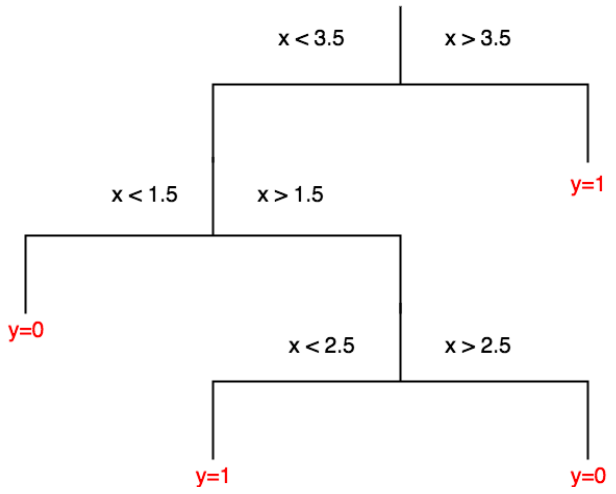


Figure 3: Decision tree learnt from a simple data set

Using decision trees in R

As you can see, while it is possible and informative to train a decision tree model by hand, this is rarely going to be practical for large data sets. It is important that you follow the examples above as these illustrate the fundamental procedures that any decision tree training algorithm follows, but we will also need to see how to efficiently train trees for larger and more complex problems using R.

Lets see how this works by investigating a classification problem that logistic regression, the algorithm we learnt last week, would perform poorly on. Recall that an important feature of logistic regression is the assumption that the probability for the output to be a '1' rather than a '0' increases or decreases *monotonically* as one of the inputs is increased. By contrast, decision trees are good at *partitioning* the input space into distinct areas that need not be monotonic, but they are less good at capturing gradually changing functions. So lets look at a function that is neatly partitioned:

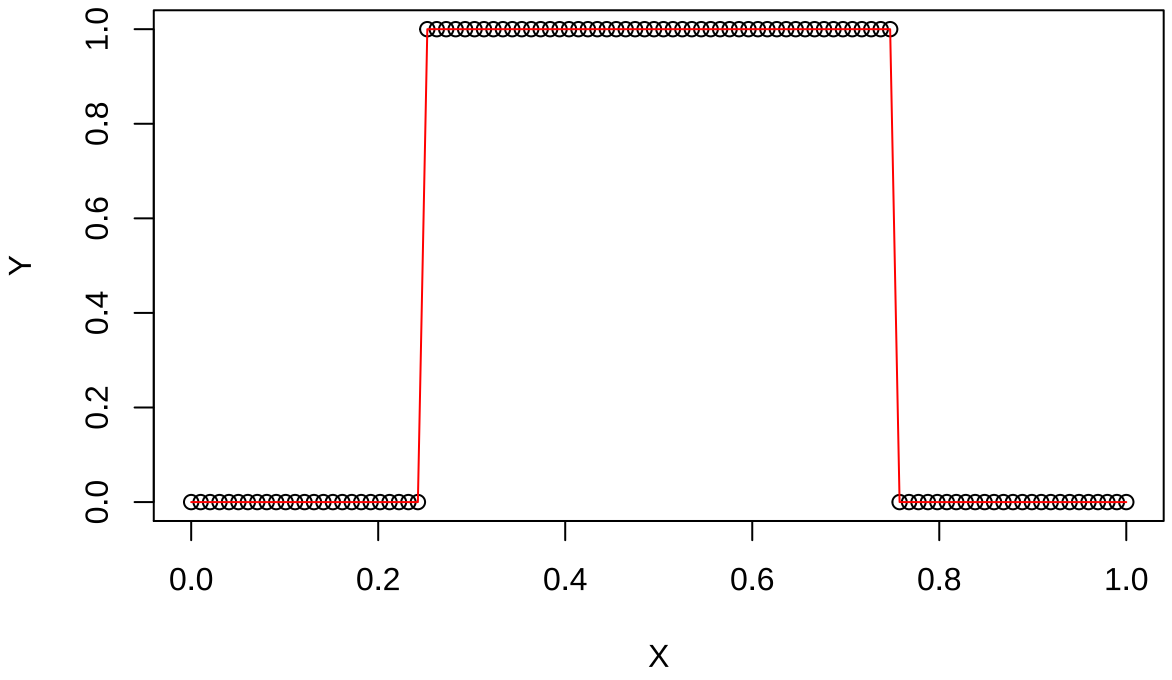
$$P(Y = 1) = \begin{cases} 1, & \text{if } 0.25 < X < 0.75 \\ 0, & \text{otherwise,} \end{cases} \tag{2}$$

that is, Y is always equal to 1 when X is between 0.25 and 0.75, and zero otherwise: a 'top hat' function. This sort of relationship cannot readily be captured by a logistic regression model (unless we use a polynomial or other basis function expansion for our input), but seems well suited to a decision tree. Lets see how we fit such a tree in R. First, we need to generate some training data from the model:

```r
#define the probability function
myfunction <- function(x){0+1*(x>0.25 & x < 0.75)}

#Generate some data
X = seq(0,1, length.out=100)
P = myfunction(X)
Y = as.numeric(runif(100)<P)
mydata = data.frame(X=X, Y=Y)

#Plot the data with the actual probability
plot(X, Y)
lines(X, P, col='red')
```



To fit a decision tree we need to use a package called **rpart**: R partition, so install this package if you haven't already:

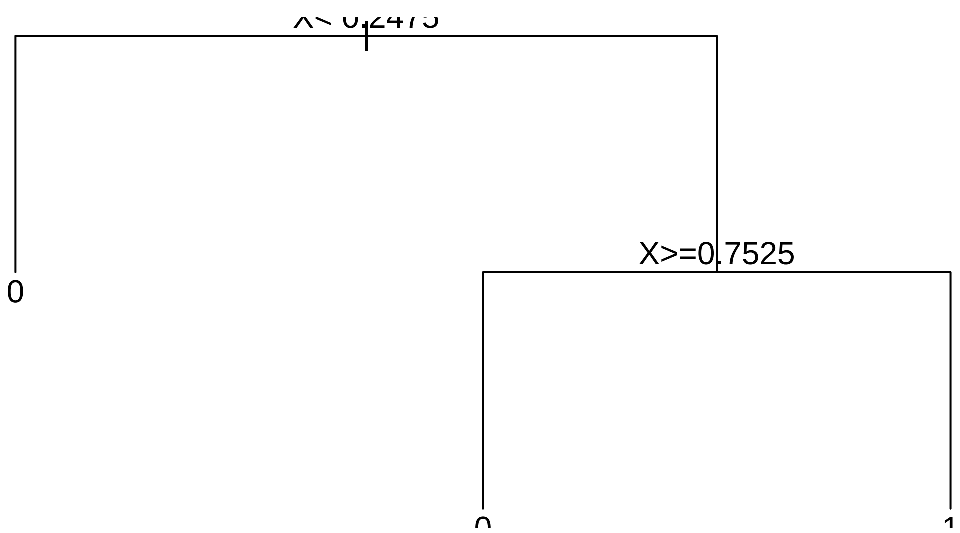
```
install.packages('rpart')
```

The package name comes from the way a decision tree *partitions* the data. Inside the package there is a function also called **rpart** that will fit the tree for us. Similarly to our use of **glm**, we need to provide a *formula* that specifies which outputs and inputs to use from the supplied data, like this:

```
#Training a decision tree with formula Y ~ X using mydata
library(rpart)
mytree = rpart(Y ~ X, data=mydata, method='class')
```

We can use **plot** to see what the tree looks like. The **text** command adds labels to each decision:

```
plot(mytree)
text(mytree)
```

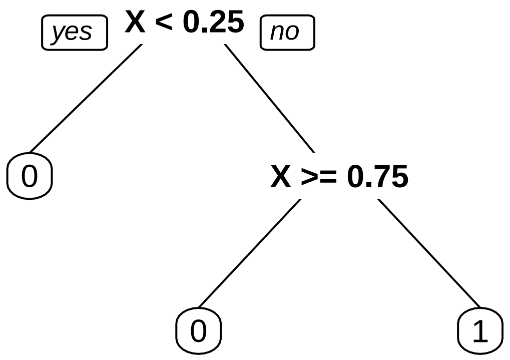


As you can see, this basic way to plot the tree is quite ugly!! We can use some improved tools to get a better visualisation using another package called **rpart.plot**. First, if needed, install the package:

```
install.packages('rpart.plot')
```

Now visualise the tree using the function **prp**:

```
#Visualising a decision tree
library(rpart.plot)
prp(mytree)
```

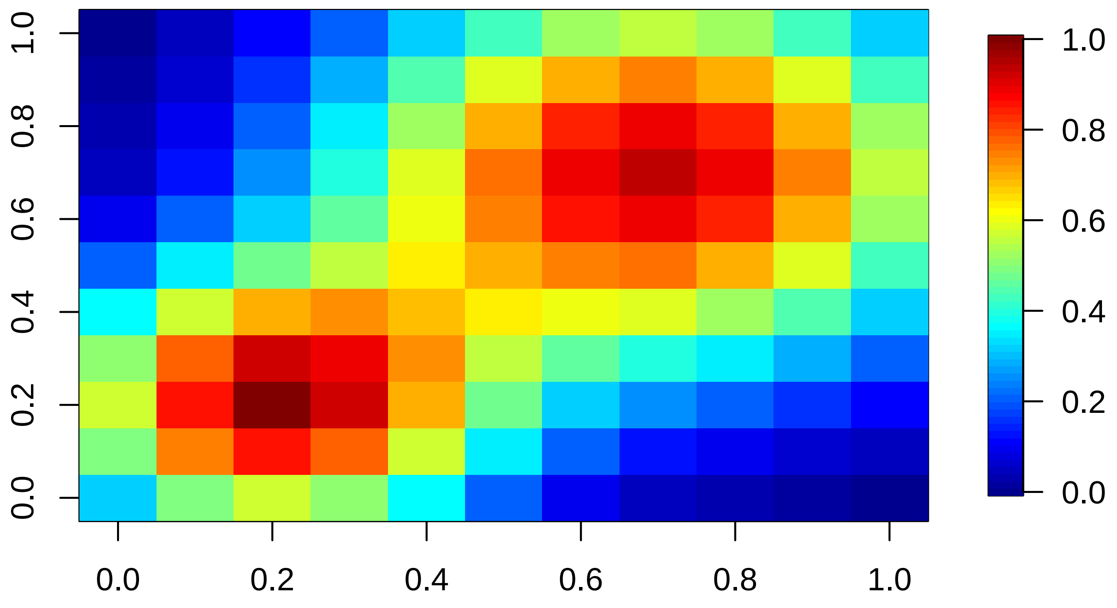


Now that looks much clearer. Looking at the tree we can see it has captured the function we generated the data from perfectly. First we check whether X is less less than 0.25. If it is then we know that Y is zero. If X is greater than 0.25 we check whether X is greater than 0.75 – if so then Y is again zero. However, if X is greater than 0.25 and less than 0.75 then we know that Y is one.

What about more complex problems? Lets see how well a decision tree model performs on a task with more inputs, and where the ground truth is not so clear cut as in the example above. Lets create a probability field that varies in two dimensions (x and y) and takes the form of several 'blobs', and generate discrete binary observations using this field. First the probability: I'll form this by taking the sum of a few 2D Gaussian blobs with different centres – you don't need to follow all the code below, but pay attention to the plot of the resulting probability field.

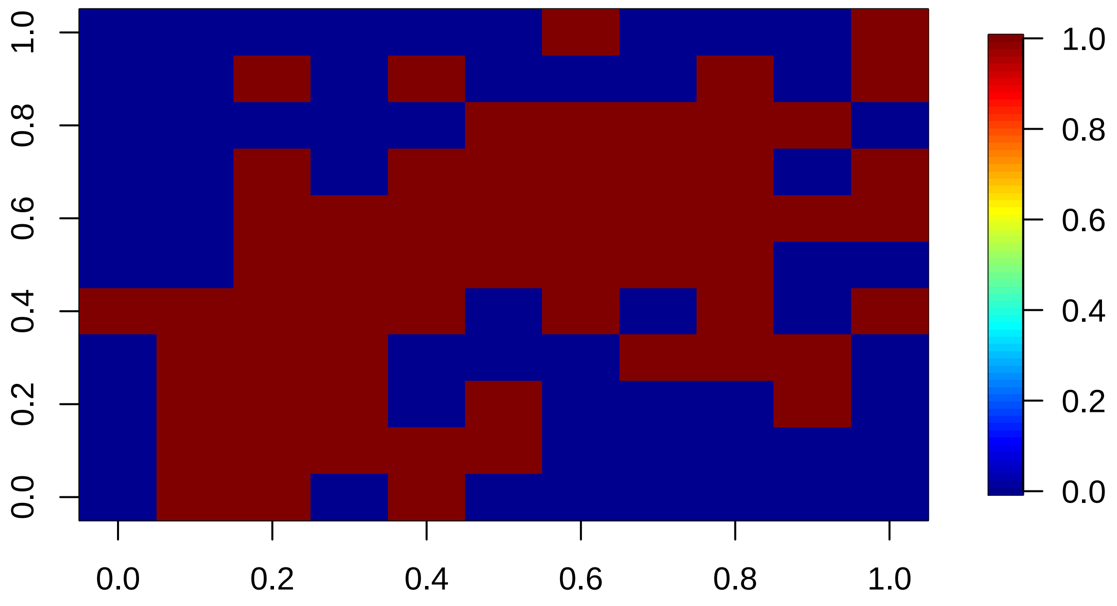
```
#create x and y cordinates on a grid of spacing 0.1
X = rep(seq(0, 1, 0.1), times = 11)
Y = rep(seq(0, 1, 0.1), each = 11)
```

```r
d1 = sqrt((X-0.2)^2 + (Y-0.2)^2)/0.2
d2 = sqrt((X-0.7)^2 + (Y-0.7)^2)/0.3
#define the probability
P = dnorm(d1) + dnorm(d2)
P = P-min(P)
P = P/max(P)
fields::image.plot(matrix(P, 11,11))
```



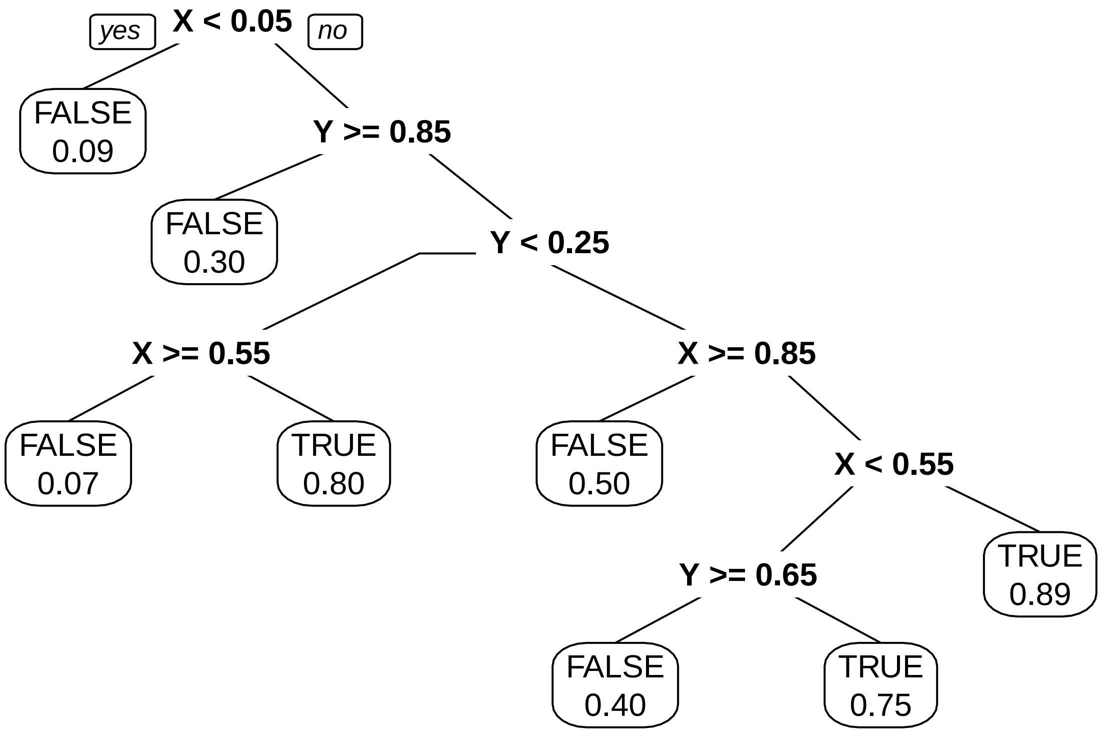
Now for each point on the grid I will generate an observation, Z, of either 1 or 0, based on these probabilities. See how noisy the data looks compared to the actual probability underneath.

```r
Z = (runif(121) < P)
fields::image.plot(matrix(Z, 11,11))
```



Now we try to learn a decision tree from this data, to see if we can recover the initial probability field. First we fit the model and visualise the resulting tree

```
#Form a data frame
mydata = data.frame(X=X, Y=Y, Z=Z)
#Fit the decision tree model
mytree = rpart(Z~X+Y, data=mydata, method='class')
#Visualise the tree
rpart.plot::prp(mytree, extra=6)
```

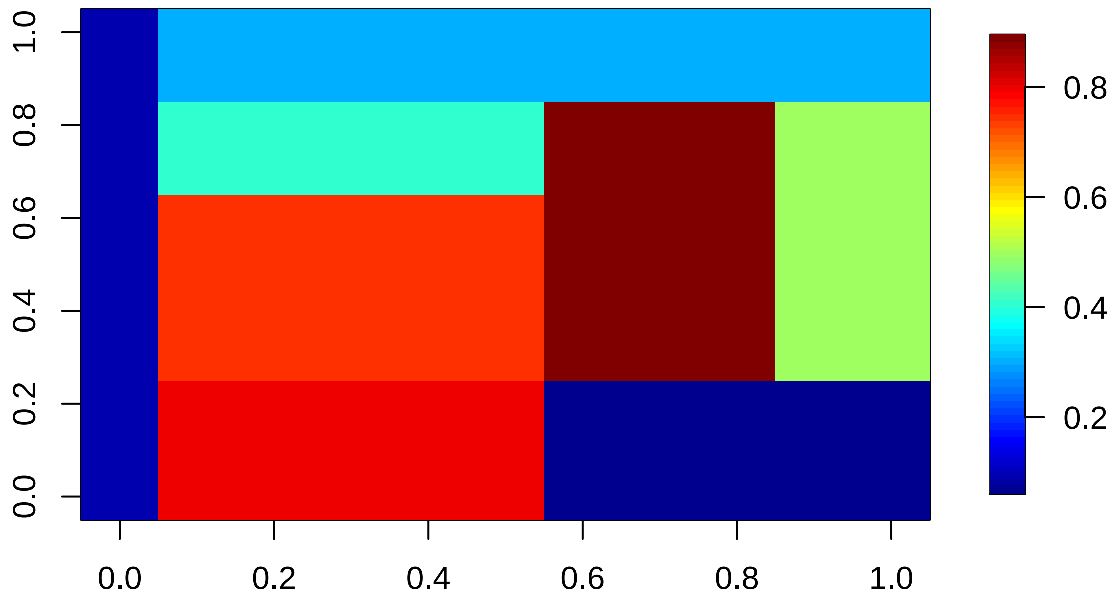


Notice two things. Firstly, the resulting tree is a lot more complex than those we've seen previously – there are a lot more decision points. This is a natural consequence of the problem becoming harder, but it means that the model is increasingly difficult to interpret just by looking at the tree. Secondly, by using the option **extra=6**, I've added the class probability to each end point. This is the proportion of cases at this terminal that have Z=1 or Z is TRUE. In the previous examples we ran the fitting algorithm until every terminal node was 'pure', i.e. it contained only one class. In practice, for more complex problems, the fitting algorithm usually stops before this point to prevent overfitting. There are adjustable options within the function **rpart** that you can try adjusting to change this behaviour: try the **cp** option within **control** to see how the tree can be made more or less complex.

Since the tree is now more complex, we can get a better idea of what the fitting algorithm has learnt by plotting the predictions the tree makes in the original x and y coordinates, rather than looking at the tree itself. To do this we use the **predict** function, just like with **glm**:

```
#predict using the fitted tree
prediction= predict(mytree, newdata=mydata)
#Extract the second column, which gives the
#predicted probabilility that y is TRUE
prediction = prediction[, 2]
#plot the prediction on the grid
```

```
fields::image.plot(matrix(prediction, 11,11))
```



What do we notice? Most obviously, this does not look much like the original probability function. This is quite normal – decision trees are often quite poor classifiers by themselves. In later lectures we will look at how they can be improved. Secondly, notice the sharp partitions – this is because the decision tree cannot easily model smooth variations in the probability, as it fundamentally works by partitioning the space. Each line in the plot corresponds to a decision node in the tree.

Finally, try running the code above again a few times, generating slightly different data (due to Z depending on the random number generator). Each time, fit a tree model and look at the tree and the plot as above. You may notice that the fitted model varies quite a lot each time. This is because decision tree models are very *unstable*. Slightly different data sets can produce very different trees. This seems like a weakness, as it means we can never be very sure that the tree we fitted is the 'right' one, since a different data set might produce a completely different result. In fact, we will see in a later lecture that this apparent weakness is actually the key to unlocking the real power of decision trees, as we will use them to build not just decision trees, but decision *forests*.