Data Mining Lab 2: A Basic Tree Classifier

Learning Objectives

- How to fit a basic classification tree
- Understand the tuning parameters of trees
- Examine the output from trees
- Produce a plot of a tree

1 Examining the data

In this lab we are going to look at the Titanic data set, which provides information on the fate of passengers on the maiden voyage of the ocean liner 'Titanic'.

The data is on Blackboard. There are 4 variables `Class, Sex, Age` and `Survived`. We are interested in predicting who survived.
The first thing you should always do is spend a reasonable amount of time examining your data and using all the techniques discussed in previous labs. Obviously there are time constraints in the labs but I strongly recommend producing some simple tables. One possibility is to look at each independent variable versus the target variable `Survived`. Experiment with some more graphical and numerical summaries.

Exercise: Are there any variables that look particularly related to `survived`?

2 Fitting a Tree

We are now ready to see how well we are able to fit a classification tree in order to predict survival (dependent variable) based on passenger class, age and sex (explanatory variables). Load the `rpart` package by using the **library** command. Dive straight in and run the following:

Remember what I said about the attach command

```
> attach(Titanic)
> fit = rpart(Survived ~ Class + Age + Sex)
```

The alternative is not to attach the dataframe and type the following

```
> fit = rpart(Survived ~ Class + Age + Sex,data=Titanic)
```

The first command allows us to directly reference the column names whilst the second line tells `R` to fit a tree where Survived is to be predicted from Class, Age and Sex and then store that tree for later reference in the object called `fit`. I really would encourage you to look at the help file for **rpart**. There is also a good introduction in the form of a vignette on the CRAN website.

Try using the command `attributes(fit)` to see what the object `fit` contains. This is a general command and is very useful to see what other information is generated. To access the parameters type `fit$<name>` e.g. fit$parm

3 Interpreting the output

You can look at the tree that has been built by just looking at the `fit` object:

```
> fit
```

This will bring up an initially intimidating looking mountain of text. However, be sure to take time and study it carefully - it should start to become clear. We will look at it in class in detail. A section of the output will look like:

```
node), split, n, loss, yval, (yprob)
      * denotes terminal node
```

and this acts as a guide in identifying what each of the values in the output means. Remember also to consult the `rpart` documentation available from `CRAN` if you need additional information. The `1)` that you see in your output is the root node where you start when trying to classify a new observation. You then choose between the indented `2)` or `3)` lines – one is when sex is male, the other female. Were this a female observation you would then proceed down the tree in the same fashion until you reach a terminal node of the tree, signified by a `*` at the end of the line. The Yes/No before the bracketed numbers then indicates the prediction the tree has returned based on the information you provided. We will go over the output in detail in class.

Exercise: Try drawing the tree from this output.

If you run

```
> summary(fit)
```

Another section of the output shows up with an initial table followed by a series of summary statistics which relate to each individual node. We are not concerned at this point in delving into the specifics of each node, but the initial table containing information about the complexity parameter, rel error, xerror and xstds is very important. You should have a general understanding of what this table is showing you about the tree. As a quick reminder:

CP	=	complexity parameter as calculated in class but divided by R(0), the misclassification rate for the root node. We will look at this in class today. The default stopping value for cp = .01. We will see at a later stage what happens with more complicated data.
nsplit	=	number of terminal nodes−1 at that point.
rel error	=	relative error or misclassification for tree at that stage — to convert to absolute error multiply by root node misclassification rate.
xerror/xstd	=	refer to the results of a cross classification procedure Ideally we want to pick the tree with the lowest xerror ± 1 SD. Again these are relative errors: to convert multiply by root node misclassification rate.

Note: We will cover this output in detail in class so make sure and bring a copy of the output with you.

Do not proceed with the lab until you're happy that this text output makes *some* sense: at this stage the detail of the numbers is not particularly important, just make sure you can see how the classifier is working. So fitting a tree is remarkably easy in `R`, but:

1. the text output is not great
2. we don't want to have to manually predict the survival if we have new data.

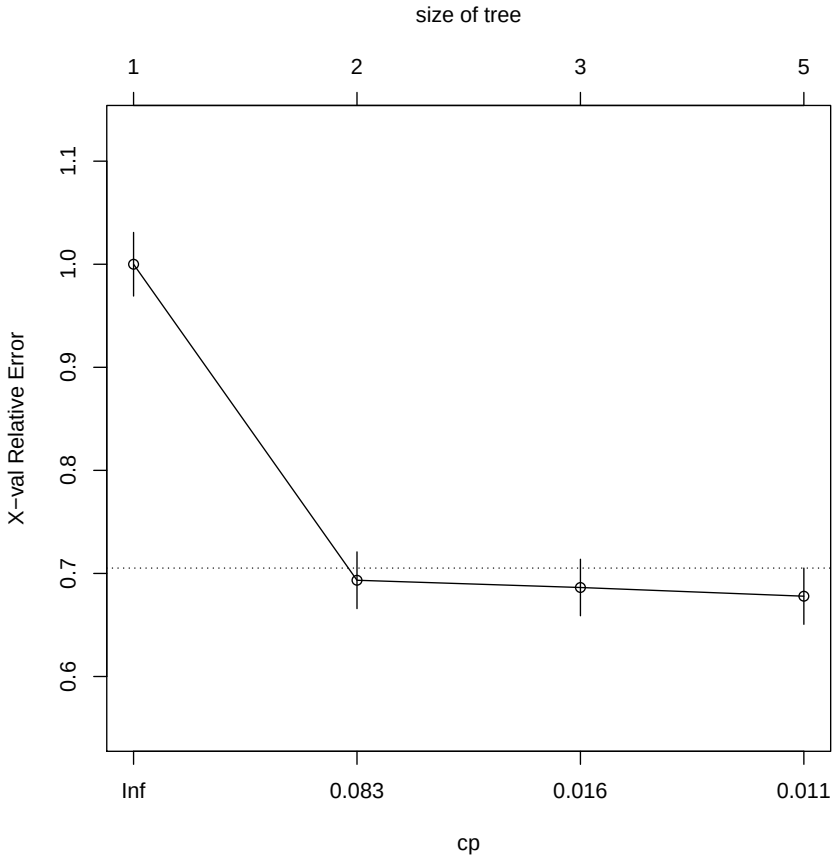
Exercise: The procedure generates more data. Try using the command `attributes(fit)` to see what the object `fit` contains. This is a general command and is very useful to see what other information is generated. To access the parameters type `fit$<name>` e.g. fit$parm
There are a number of other different output options available but the key is really to extract the meaningful information. In our basic tree model we are most interested in the complexity parameter so it is best to look at this in a bit more detail. Enter the following command

```
> printcp(fit)
```

You will see the same table that we looked at when using the `summary()` appears on its own. This command is useful when you have a large tree and don't need to print out all the additional information for each node that `summary()` provides. An even easier way to interpret the complexity parameter output is to plot its progression:

```
> plotcp(fit)
```

You should get a graph similar to the one shown below...

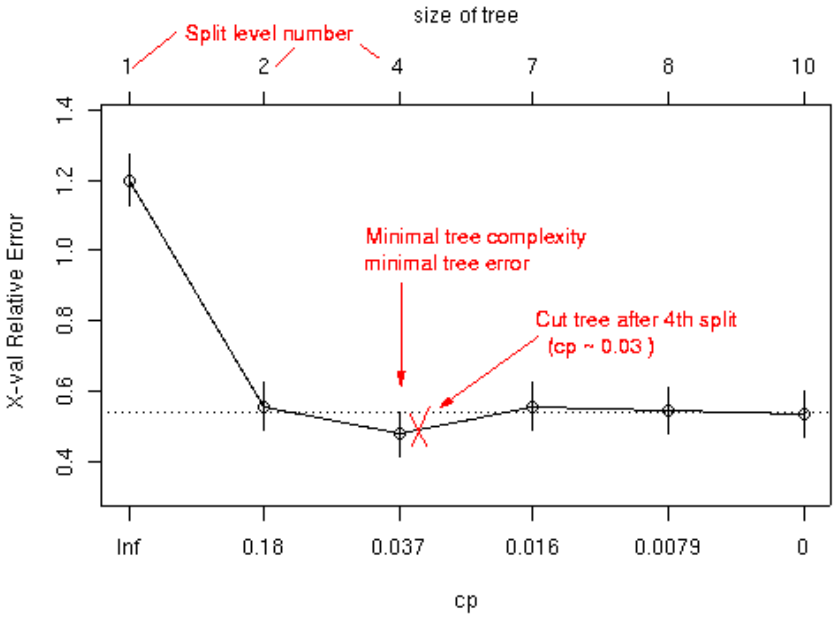


Ultimately the complexity parameter is used to control the size of the tree. Smaller values will build more complete trees (with CP = 0 building a full tree) and larger values dictating that a larger "benefit" must be achieved before growing the tree. We want to find model with a CP value that minimises the cross validation relative error (y-axis in the graph above) To do this we must adjust some parameters in our original `rpart` model. In the graph you can clearly see that the plot hits a minimum at CP = 0.011. As mentioned in class we choose a value for **cp** corresponding to min(xerror)+ 1SE. This is the dotted line on the graph. Remember this will change slightly from run to run. So we actually have quite a good model and any parameter adjustments are not likely to make any great headway.

The following is a useful piece of R code for finding this value. This assumes that the results from the model are in an object called **fit1**. Make sure you understand why this codes works.

```
i.min<-which.min(fit1$cptable[,"xerror"])
i.se<-which.min(abs(fit1$cptable[,"xerror"]
 -(fit1$cptable[i.min,"xerror"]+fit1$cptable[i.min,"xstd"])))
cp.best<-fit1$cptable[i.se,"CP"]
```

For the purposes of illustration the plot below is taken from a webpage (http://www.grassbook.org/neteler/shortcourse_grass2003/notes7.html) which does a run through of `rpart` modelling on a more complex dataset about beetle presence in Spearfish, USA (exciting stuff). You can see that the plot of the complexity parameter initally drops and then rises again - a common occurrence in models. Therefore we must prune the model so that the tree only grows to its minimum CP. You will have covered the process of pruning in more detail in lectures.



4 Tree Parameters

Despite the fact that tuning may have little effect on this model we will look at the parameters nonetheless so you can have a go on your own in the next lab! Although the code used to generate our tree model was really simple, there are actually a huge number of adjustable parameters available for use with `rpart`. You need to consult the `rpart.control` documentation available from `CRAN`, but we will look at the two most common adjustments in this lab - the splitting

criterion and the complexity parameter. `rpart` defaults to a Gini split (this is why it never appeared as a parameter in our original model), but the other option is to explicitly tell the model to split using the Information split also known as Entropy. To do this run the following code:

```
> fit = rpart(Survived ~ Class + Age + Sex,data=Titanic,
control= rpart.control(split='Information'))
```

Exercise: Generate a textual summary output and a plot of the CP against the cross validation relative error. Are there any differences in the original output (which used Gini) compared with the output generated using the Information split?

Assuming we are sticking with our original model (using the Gini split) we can run either of the two following lines knowing that the output will be the same.

```
> fit = rpart(Survived ~ Class + Age + Sex,data=Titanic)
> fit = rpart(Survived ~ Class + Age + Sex,
data=Titanic, control=rpart.control(split='Gini'))
```

Now we are going to adjust the complexity parameter of the model. Again `rpart` defaults the cp value to 0.01, hence we never saw it in our original code. To try a value of 0.05, enter the following

```
> fit = rpart(Survived ~ Class + Age + Sex, data=Titanic
               control=rpart.control(split='Gini', cp = 0.05))
```

Exercise: Can you shorten the code above? (**Hint:** Remember the `rpart` defaults)

Exercise: At this point, you can experiment with different numbers. Try some conservative values(0.001,0.003, 0.005 etc.) and then some extreme ones (1 and .000001). Examine the output in combination with `plotcp()` and see what changes occur.
To check that you are setting the parameters correctly type

```
fit$control
```

What you will find is that there is little or no movement in the tables of numbers and likewise for the plot of the CP versus the cross validation relative error. This will not always be the case and in the future you will have to work harder to get the best models.

5 Graphical Output

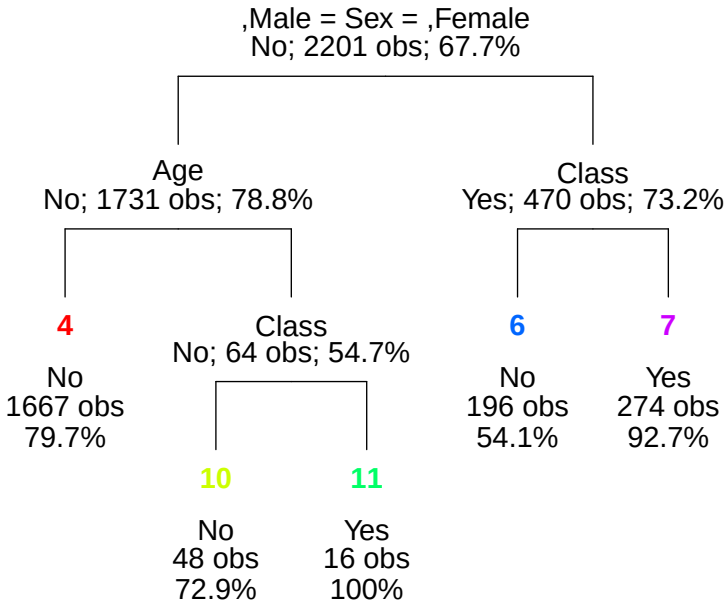
We will start just using basic plotting tools to construct a visualisation of the decision tree you have built. Run the following command to get a (very) basic tree plot:

```
> plot(fit, compress=TRUE,uniform=TRUE)
> text(fit,use.n=T,all=T,cex=.7,pretty=0,xpd=TRUE)
```

If you load the library `rattle` you can try the following which gives a more aesthetically pleasing reproduction:

```
> drawTreeNodes(fit,cex=.8,pch=11,size=4*.8, col=NULL,nodeinfo=TRUE,
> units="",cases="obs",digits=getOption("digits"),decimals=2,print.levels=TRUE,
> new=TRUE)
```

You should end up with the graph below. It is highly recommended at this stage that you look at the graph and refer back to the text based output for your `fit` model. If you can make the link between the two you are well on your way to understanding trees.



5.0.1 Improved graphical output

There is a tree plotting function called `prp` which produces much nicer graphs. The function is in the `rpart.plot` package. I have put a copy of a paper which describes the function in BlackBoard. You need to read the paper. You can do all sorts of stuff. Here are a few examples.

6 Automated Prediction

Prediction is in fact relatively easy and most of the work lies in preparing the new data we want to predict. First, we must have our query data in a new data frame, with the appropriate labels (Class, Age, Sex) to match the labels of the original data on the input/independent variables (i.e no Survived – that's what we want to predict!)

```
> newdata = data.frame(Class=c("2nd"), Age=c("Child"), Sex=c("Male"))
> newdata
```

Then, we call the aptly named `predict()` function and provide it with the tree to use (`fit`) and the data to predict (`newdata`)

```
> predict(fit, newdata)
```

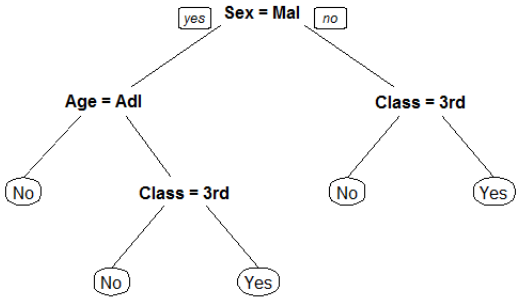


Figure 1: prp(fit)

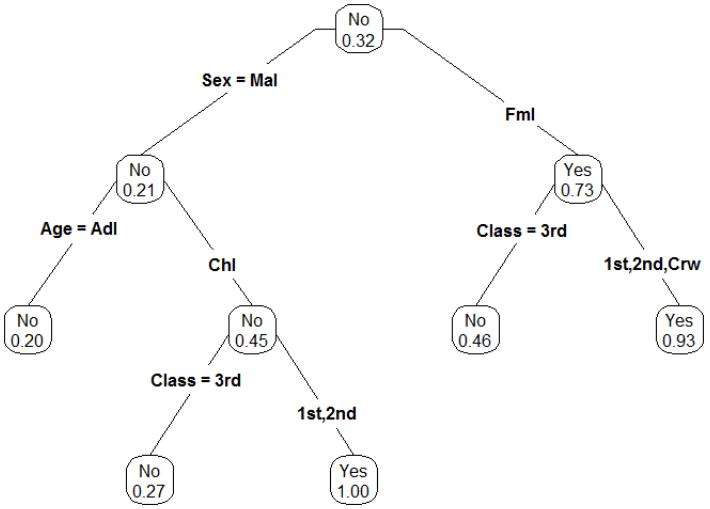


Figure 2: prp(fit,type=4,extra=6)

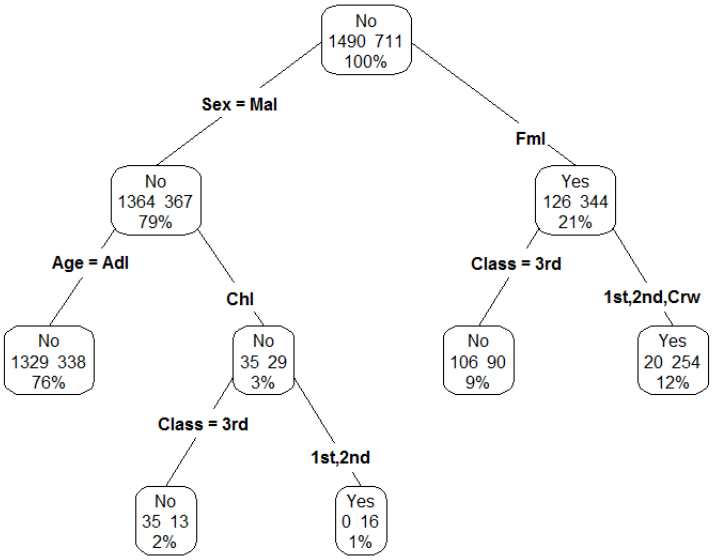


Figure 3: prp(fit,type=4,extra=101)

You will see it returns a No/Yes matrix. The number under each represents the probability it assigns to that label being true for the new observation (so here you should see it predicts Yes with apparent certainty). You may have noticed that in the `data.frame` command we declared each variable a vector by housing it in a `c()` with double quotes (`" "`) around the text. We can in fact pass in multiple new observations by adding more quoted elements to the vector and predict them all at once.

Exercise: Determine the predictions for the following by setting up a data frame with multiple observations and passing it to `predict()` along with our fitted tree model

- Female child in first class
- Adult male in second class

- Adult crewman

7 How Did The Tree Do?

A simple way to answer the question of how accurate the tree was is to simply run `predict()` on the original data and compare it to the truth. This is not quite statistically sound in-so-far as you're always going to do as well as possible on the data you fitted from and one should really test on data the fitting algorithm never saw. However, it is sufficient for a quick assessment - after all if it *can't* predict the data which trained it, then it really is doing badly! We will see in later labs how to get around this problem. But for now can you come with some simple graphs or tables to look at this.