Data Mining Lab 3: More trees

1 Introduction

In this lab we are going to look at all the data set relating to the so-called 'churn' of customers of a telephone company.WE used a small version of this dataset in lab1. We are interested in whether we can predict which customers are likely to leave (churn) to another company — this could be important to business decisions such as targeting special offers at these customers in an effort to retain them.

If you need to refer to previous labs or to download the data set, they will be in the `get folder` in `ST4003`

2 Telecom Churn Data

2.1 Getting Setup

Exercise: Load the telecom churn data into a dataframe called `churndat` in your workspace. Remember to use the `attach` function.
Exercise: Load the `rpart` package, which contains the tree building functions.

```
> library(rpart)
```

2.2 Explore the Data

Since we're looking at a new data set it's good practice to get a feel for what we're looking at.

```
> names(churndat)
> summary(churndat)
```

These two commands tell us respectively the names of the variables and some summary statistics about the data contained therein.

You've already seen how to do box plots in lab 1, so we'll look at a grid of histograms with this data instead. See if you can figure out what the following commands will do, then run them:

```
> par(mfrow=c(4, 4))
> for(i in 4:17) { hist(churndat[,i], xlab=names(churndat)[i],
                       main=names(churndat)[i]) }
```

The first command tells R that we want a 4×4 grid of plots on the same screen, while the second command creates a loop which plots the 4^{th} to the 17^{th} variable in histogram format. Note that we don't do histograms for the first three variables as they are not numeric or in one case is an phone area code and not as distributionally interesting. Try adapting this to look at a series of appropriate box plots. You may want to change the layout with just two plots per column.
Exercise: Which variable is the best for discriminating between those who churn and those who do not churn?

2.3 Train and Test Subsets

You have now seen in lectures the importance of separating the data into train, validate and test sets to ensure that you don't bias the fitting and generalisability of your tree. Thus far in the labs we've 'cheated' and ignored this, but for this new data set we'll be more rigorous.

The first thing to do is to split the data in `churndat` into two new variables, `train` and `test` (we don't need to explicitly create a validate set as the `rpart()` function handles this internally).

```
> test_rows = sample.int(nrow(churndat), nrow(churndat)/3)
> test = churndat[test_rows,]
> train = churndat[-test_rows,]
```

The first line will select a random sample of one third of the numbers between 1 and 3333 (the number of observations). The second line will select those row numbers and place them in the `test` variable. The third line takes everything *except* those row numbers and places them in the `train` variable.

Thus, we now have a $\frac{2}{3} : \frac{1}{3}$ split of the full data in `train`:`test`. To get the same split each time use `set.seed x` where x is any number. To get the same split use the same number `x`.
Exercise: Check to see the number of cases for each class in the variable `churn` in each of the `train`and `test` datasets. Hint: use the `table` function.

2.4 Constructing the Tree

We now want to build the classification tree (obviously on the `train` data). However, we are not going to leave R to choose the complexity parameter this time.

```
> fit = rpart(churn ~ ., data=train[,c(3:18)], parms=list(split="gini"),
              control=rpart.control(cp=0.0001))
> summary(fit)
```

We use the c(3:18) to select out variables we want to include in model as `churn .` will use all the variables. You will get a huge wall of text! If you scroll up to the top you'll see the table listing the complexity parameter along with the various splits and errors. Notice how the xerror column now decreases and then starts to *increase*.

Just to refresh your memory

CP	=	complexity parameter as calculated in class but divided by R(0), the misclassification for root node. You should know how to reproduce this from the lectures. The default stopping value for cp = .01. We will see later 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 error.
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 error.
Impurity	=	decrease in impurity$\times n$. This is why it is such a big number.

Exercise: What is the best value for the complexity parameter on the basis of this table?
Exercise: Refit the tree, this time using the best value for the complexity parameter (to two significant figures – this is so we round and don't overshoot) and store it in the variable `fit`.

The tree we've now fitted is rather too large to plot nicely, so we'll just look at the simple text version:

```
> fit
```

Exercise: Rerun the last two commands, this time changing the split criterion in the first to `"information"` instead of `"gini"`. The `information` splitting criteria is equivalent to the Entropy as discussed in class. Is there a change in the tree?

2.5 More Tree Detail

We want to continue to dig deeper into understanding the tree than in the last lab.

Before the detail, we can get a nice summary of the terminal node rules with the following:

```
> library(rattle)
> asRules(fit)
```

This produces a description for each node as well as the following:

```
 yval=Yes cover=108 (5%) prob=0.88
```

Thus means that this node is assigned to class Yes, the cover gives the number of observations, the % is the of the total number of observations and the prob is the prob of Yes for that node.

3 Missing Data

The Churn data set is very nicely behaved in so far as having no missing data at all. However, many real world data sets you are likely to encounter will have missing data.

We are going to intentionally 'mess up' the data to see what happens. We are going to allocate randomly missing data to a couple of variables. Start by choosing the most

important variable - let us call it `var1` and allocate a random sample of 10% of the cases to missing e.g. NA. In the following commands you have to substitute the variable name for `var1`

```
> var1[seq(1,3333,10)] = NA
```

Double check that this worked by using `summary`. Be careful that you know what varaible you are changing. .

As covered in lectures, trees handle 'missingness' by using other non-missing variable values as surrogates to predict the absent value. In `R` we are able to specify how many other variables can be used as surrogates as follows (note that the following is all on one line in `R`): We can set the number of surrogates we wish to look for as follows: Suggest you try three and see what happens

```
> fit = rpart(churn ~ ., data=train[,c(3:18)], parms=list(split="gini"),
              control=rpart.control(cp=0.0001,maxsurrogates=3))
```

Exercise: Has the tree changed much with the missing values? Explain the surrogate results.

4 How good is the tree?

if you have any time left you might like to calculate a confusion matrix using the `test` data set.