Data Mining Lab 4: New Tree Data Set and Loss Matrices

1 Introduction

In this lab we are going to look at a new data set relating to the so-called 'churn' of customers of a telephone company. 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 on the course labs website:

`http://www.maths.tcd.ie/~louis/DataMining/`

2 Telecom Churn Data

2.1 Getting Setup

Exercise: Download the telecom churn data from the course website.
Exercise: Load the telecom churn data into a variable called `churn` in your workspace. (Hint: see lab 2, §2.1 if you've forgotten)

```
>
```

Exercise: Load the `rpart` package, which contains the tree building functions. (Hint: see lab 2, §3.1 if you've forgotten)

```
>
```

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(churn)
> summary(churn)
```

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 §5.3, 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(churn[,i], xlab=names(churn)[i],
                        main=names(churn)[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.

At a glance, most of these histograms have a gratifyingly Normal distribution shape. It is interesting also to note that `vmail.msgs` looks very zero-inflated (since many people don't have voice mail) and `intl.calls` appears skewed.

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

Exercise: Run the `summary()` function on `train$churn` and `test$churn` to look at how the random split has dealt out churn/non-churn lines:

```
>
>
```

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, parms=list(split="gini"),
              control=rpart.control(cp=0.0001))
> summary(fit)
```

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*.

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
```

2.5 Loss Matrix

The next goal is to incorporate a loss matrix in to the classification tree: these are a parallel to the profit matrix from lectures. A loss matrix is used to weight misclassifications differently. In other words, different problems may mean that a false positive (type I error) and false negative (type II error) are not equally bad. For example, quite often in medical screening a false negative is far worse (we miss that the person has a disease) than a false positive (we suspect they have it when they don't and send for further testing). Our classification tree can take this into consideration by weighting how much to penalise each incorrect classification in a given choice of split.

$$L = \begin{pmatrix} 0 & L_{fp} \\ L_{fn} & 0 \end{pmatrix}$$

Note: due to the quirk of `R` *ordering factors alphabetically, you need to be careful about the order of false positive and false negative in the above matrix. Here, our churn factor is yes/no, so no is first ('n' being before 'y') and so the first row/column is actual/predicted for* ***no****.*

2.5.1 Fitting with a loss matrix

Run the following code, replacing `***` with the complexity parameter value you found earlier.

```
> fit = rpart(churn ~ ., data=train, parms=list(split="gini",
              loss=matrix(c(0,12,16,0), byrow=TRUE, nrow=2)),
              control=rpart.control(cp=***))
> fit
```

Exercise: Can you see the tree has changed compared to earlier?

So, this is the loss matrix we used:

$$L = \begin{pmatrix} 0 & 12 \\ 16 & 0 \end{pmatrix}$$

where we penalise false negatives more strongly than false positives. It's really important to understand these different types of error: for example, one might imagine here that the phone company is interested in reducing churn by making special offers to people who they predict will churn. In this context, they might work out that they lose more money (16) when they fail to catch someone who *will* churn (false negative) than they will lose by giving a special offer and reducing profit (12) to someone who wasn't going to leave anyway (false positive).

We can now work out the expected loss as follows:

```
> sum(c(0,12) %*% t(predict(fit, subset(test, test$churn=="No"), type="prob")))
> sum(c(16,0) %*% t(predict(fit, subset(test, test$churn=="Yes"), type="prob")))
```

Take a few minutes to try and decipher those two lines. Use the manual pages and reduce the parts into smaller chunks to get a feel for it (Hint: `type="prob"` gets us probabilities instead of just plain yes/no predictions).

In mathematical notation, they are basically doing the following two calculations:

$$\sum (\mathbb{P}(\text{customer doesn't churn}) \times 0 + \mathbb{P}(\text{customer does churn}) \times 12) \quad \ldots \textit{for the } \texttt{test} \textit{ No's}$$

$$\sum (\mathbb{P}(\text{customer doesn't churn}) \times 16 + \mathbb{P}(\text{customer does churn}) \times 0) \quad \ldots \textit{for the } \texttt{test} \textit{ Yes's}$$

2.5.2 Taking loss futher

`R` only uses the loss matrix to adjust the way splits are chosen. However, the loss matrix can also be used to tune the threshold we use on the probability of classification.

What does that mean? Well, recall that the tree actually gives us a probability of each class (based on frequencies) when we reach a terminal node. By default, we take this to mean that < 0.5 is one class and > 0.5 is the other, but we could change that to being < 0.4 is one class and > 0.4 is the other (say) if that reduces our perceived loss when making an incorrect choice (type I/II error).

We are going to create a graph showing the actual loss as we vary the probability at which we split. Don't worry about understanding the coding in this bit, it's a little beyond the scope of this lab ... just type carefully! However, you *should* understand at an overview level what the plot represents and be able to articulate how it would be calculated. *Note: the third line could take quite a while to run depending on your computer's speed.*

```
> calcLoss = function(p) { sum(c(0,12) %*% t(predict(fit, subset(test,
                           test$churn=="No"), type="prob")>p)) + sum(c(16,0)
                           %*% t(predict(fit, subset(test, test$churn=="Yes"),
                           type="prob")<p)) }
> p = seq(0,1,length.out=100)
> loss = sapply(p, calcLoss)
> plot(p, loss, type="l")
```

You should notice how the loss bottoms out over quite a broad range, though the exact detail could vary slightly for everyone because each person will have a slightly different train/test split from §2.3.

Exercise: Approximately what probability would you use as the threshold between classes for the loss matrix which we defined?

You can look at `p` and `loss` to get exact numbers:

```
> p
> loss
```

The first value in the `loss` vector is the loss at the corresponding first value in `p` and so on. Rather than trying to count through and see the minimum and match it up to a p value, we can find it more quickly:

```
> p[which.min(loss)]
```

This picks out of `p` the corresponding element which was minimum in `loss`.

Exercise: If you have time, copy and paste your code for §2.5.1 and §2.5.2, but replace every instance of 16 with 500. This creates a really huge penalty on false negatives compared to false positives: what has changed? Is this still a reasonable analysis?