Data Mining Lab 4: More Programming Concepts and Model Evaluation

1 Introduction

In this lab we are going

- Expand our repertoire of `R` programming skills.
- Create confusion tables.
- Examine how to incorporate a cost matrix.
- Draw ROC curves and other graphs covered in class.

2 Further Programming

In the very first lab we refreshed the bare basics of `R`, but didn't delve into any of the programmatic power it has. Thus far, we have basically given one-line-at-a-time instructions to be run (with some exceptions which you were not expected to follow at the time). Now we will look at control structures and functions which will enable you to build more powerful solutions.

2.1 Control Structures: 'if'

The `if` statement provides a simple way to perform different actions depending upon whether some condition you specify is true or not. The basic structure is as follows:

```
if(<condition>) {
    <do this if condition evaluates to TRUE>
} else {
    <do this if condition evaluates to FALSE>
}
```

For example, the following generates a Normal($\mu = 0$, $\sigma^2 = 1$) random number and prints to the screen which side of 0.25 it lies:

```
i = rnorm(1)
i
if(i > 0.25) {
    cat("i is greater than a quarter")
} else {
    cat("i is less than or equal to a quarter")
}
```

Exercise: Use `R` to simulate a fair coin toss. Your code should simply print "Heads" or "Tails" to the screen once every time you run it, each time with a probability $p = \frac{1}{2}$ for a head or a tail.

2.2 Control Structures: 'for'

A frequent issue you will face when doing any statistical coding is the desire to perform the same action repeatedly many times. To avoid having to copy-and-paste the section of code to repeat it (eg if we want to do something 1000 times!), we can simply use loops which will perform an action as often as we specify. The `for` loop is the simplest of these (there are also `while` and `repeat` loops). The structure is:

```
for(<variable> in <vector>) {
    <do this, one time for each element in the vector, setting variable
     to the value of that element each time>
}
```

For example, the following repeatedly calculates the log of a value and prints it. In this case, we are evaluating the log of the numbers 1 to a set value (10 here). Notice we also keep track of the sum of all the calculations to date and print this after the loop.

```
total = 0
upto = 10
for(j in 1:upto) {
    total = total + log(j)
    cat("The natural log of", j, "is", log(j), "\n")
}
cat("The sum of the natural logs of 1 to", upto, "is", total)
```

Be sure to input and run this and ensure you understand it completely. Note:

- that the `"\n"` simply creates a new line so that the next output is on the next line.
- we could have typed `1:10` instead of `1:upto`, but this way we can change one thing and all the results adjust.
- here we used the variable `j` which had the value from each element of the vector we looped over. However, you don't <u>have</u> to ... if you ignore it you just get an identical section of code run as many times as there are elements in your vector.

Exercise: Using a for loop, simulate 100 tosses of a fair coin and count how many heads you get in a variable called `total` which you print out after the loop.

2.3 Functions

Functions allow us to bundle up code into easy to use commands that we can use without having to think about what's going on inside. We've used many many functions on the course so far, like `rpart()`, which were written by the creators of `R`. However, things which are specific to our own work can be made much easier if we also create functions rather than having reams of code in a file. Combining loops and functions can also give us powerful abstractions that make statistical work very easy. The structure is:

```
<name of function> = function(<arg1>, <arg2>) {
    <do things here, with the variables arg1 and arg2 available for use>
    <this last line is what the function will return as the result>
}
```

and it is called like so

```
<result> = <name of function>(<variable to pass 1>, <variable to pass 2>)
```

This kind of meta-definition is getting horrible looking, but an example should clarify! The following draws the specified number of cards from a deck of 52 with replacement:

```
drawCards = function(number) {
    faceValues = c(1:10, "Jack", "Queen", "King")
    suits = c("Hearts", "Diamonds", "Spades", "Clubs")
    f = sample(faceValues, number, replace=TRUE)
    s = sample(suits, number, replace=TRUE)
    paste(f, "of", s)
}

drawCards(5)
```

Note:

- `sample(v, n, replace=TRUE)` draws `n` elements with replacement from a vector `v` with equally likely probabilities.
- `paste()` combines all of the elements of it's input into a single string, element by element.

So now, anytime I ever want to do something in `R` which involves drawing cards from a deck, all I need to remember and use is `drawCards()` and I don't clutter my code with the rest!
Exercise: Create a function called `tossCoin` which simulates coin tosses. It should take a single argument specifying how many coins to toss, and should return just the total number of heads from the simulation.

3 Telecom Churn Model Evaluation

We return now to the world of data mining and will put the above control structures and functions theory to use in evaluating the classification tree model of the telecom churn data.

3.1 Getting Setup

Exercise: Load the telecom churn data into a variable called `churn` in your workspace. The previous labs are available if you need them..At this stage you should have a script file to work from.
Exercise: Load the `rpart` package, which contains the tree building functions.

Now run the following code which will create our train/test split; fit a tree on the train data; and then calculate the "Yes" prediction probabilities for the test data.

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

> fit = rpart(churn ~ ., data=train[,2:18], parms=list(split="gini"))
> fit

> probs = predict(fit, test)[,2]
> probs
```

Again remember that the expression `train[,2:18]` selects out variables in columns 2 to 18 of the data set. We do not want to include variable 1 `acct` as it is just the account number.

3.2 Goal

The goal today is to evaluate the model which has been fit by R. Remember the cut-off probability (α) for splitting Yes/No is the primary thing over which we have control *after* the model is fit (we have CP and loss matrices specified *before*) and so that plays a key role in the two things we look at today: Confusion Charts (at different α) and Receiver Operating Curves (a curve created by varying α). R has functions to create both these, but as an exercise we will create them ourselves.

3.3 Loss Matrix

The next goal is to incorporate a loss matrix in to the classification tree. 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**.*

3.3.1 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).

3.4 Confusion Charts

Confusion charts can help us see whether our classifier is making false positive or false negative errors more frequently for a given α. We will need a function which takes (i) our prediction probabilities for yes ($\mathbb{P}[\text{churn} = \text{Yes} \,|\, \mathbf{x}]$); (ii) our chosen cut-off α; and which then returns actual Yes/No predictions. Below is a template with one omission.

```
makePrediction = function(probYes, cutoff) {
    prediction = vector(length=length(probYes))
    for(i in 1:length(prediction)) {
        if(???) {
            prediction[i] = "Yes"
        } else {
            prediction[i] = "No"
        }
    }
    factor(prediction, levels=c("No", "Yes"))
}
```

So, `probYes` will be a vector of probabilities and `cutoff` will be a number between 0 and 1 which is α.

Exercise: Understand what is going on and so figure out what `???` on the fourth line should be.

Now we can compute the confusion table for any α we choose. Here's for $\alpha = 0.4$ as an example:

```
> pred = makePrediction(probs, 0.4)
> table(pred, test$churn, dnn=c("Predicted", "Actual"))
```

3.5 Drawing ROC and other curves

We use the package `ROCR` to draw these curves. First load the package and also the file ROCR.pdf You should keep a copy of this file as it explains how the package works.

We first fit the model like above and assign it to a variable called `fit`. You can call it anything you like. We then create the predicted values on the test dataset as follows:

```
>predp=predict(fit,newdata=test)
```

This creates a 2 column vector of probabilities. The first probability corresponds to the probability of churn = ”NO” and second is the probability of churn = ”YES”. the columns are ordered either numerically or alphabetically. Try typing `pred` to see what it contains. Next we create a prediction object as follows:

```
>predics=prediction(predp[,2],test$churn)
```

To see what is in `predics` type

```
>str(predics)
```

You can list the `fp` values by typing

```
>predics@fp
```

To draw an ROC curve you first choose what measure you want on each axis . In an ROC curve we want "tpr" on Y-axis and "fpr" on x=axis. . The default value for the x- axis is the cutoff. See the file `ROCR.pdf` for a list of the measures you can use.

```
>perfo=performance(predics,"tpr","fpr")
>plot(perfo)
```

The following command plots accuracy vs cutoff

```
>perfo=performance(predics,"acc")
>plot(perfo)
```

Check what the object `perfo` contains by using `str`. You can also add different subcommands to the plot command. See the file `usingR.pdf`

3.6 ROC

We can also create an ROC curve by creating a function. This is another example of R programming. Previously we created a function to calculate the predictions for any given α, we can now use this from within *another* function which calculates it over a range in order to calculate and draw a ROC. We will design our function to take the model we fitted and the test set, then draw the ROC for it by computing the true and false positive rates over a range of 100 values between 0 and 1.

```
drawROC = function(fit, test) {
    alpha = seq(0, 1, length.out=100)

    tp = vector(length=100)
    fp = vector(length=100)

    probs = predict(fit, test)[,2]
    for(i in 1:100) {
        pred = makePrediction(probs, ???)
        confusion = table(pred, test$churn, dnn=c("Predicted", "Actual"))
        tp[i] = confusion[2,2]/(confusion[1,2]+confusion[2,2])
        fp[i] = confusion[2,1]/(confusion[1,1]+confusion[2,1])
    }

    plot(???, ???, type="l")
}
```

Exercise: Understand what is going on and so figure out what the three `???`'s should be.

Once you've done that, run the following to see the ROC for the model we fitted to the train data in §3.1

```
> drawROC(fit, test)
```