Data Analytics Lab 4: Model Evaluation I

Learning Objectives

- Learn how and why we split data into training and test sets
- Build prediction objects and use prediction values
- Begin using the package `ROCR`
- Create boxplots and confusion tables for basic model evaluation
- Creating Confusion Tables with a great example of an `R` function.

In this lab (and the next) we will use our previous tree models to conduct an evaluation of the performance of the models. It is essential that you understand the techniques applied in these labs as evaluation is key not just to trees, but to all the other data mining techniques applied in this course.

1 Getting set up

Exercise: Load the churn data into a data frame called `churndata` in your workspace. At this stage you should have a well documented script file to work from. It is advisable to start with `summary(churndata)` just to make sure that you have the right dataset and variables.

2 Training and test sets

You have now seen in lectures the importance of separating the data into train and test sets to ensure that you obtain an accurate estimate of the generalisability of the tree. Thus far in the labs we've 'cheated' and ignored this, but for this new data set we'll be more rigorous. We can use the `set.seed` function in `R` to manually set a seed value for the split. If you use the same seed you will get the same split. For example:

```
> set.seed ( 58)
```

The first thing to do is to split the data in `churndata` into two new dataframes, `train` and `test`. One way to do this is as follows:

```
> test_rows <- sample.int(nrow(churndata), nrow(churndata)/3)
> test <- churndata[test_rows,]
> train <- churndata[-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` dataframe. The third line takes everything *except* those row numbers and places them in the `train` dataframe. Thus, we now have a $\frac{2}{3} : \frac{1}{3}$ split of the full data in `train:test`. You should be able to adjust this to split in any proportion.

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. You can of course look at all the variables as well.

3 Using the training and test sets

3.1 Use the training set to build the model

Exercise: Given that we have now split the data we must adjust our original tree model that we built in the last labs. Build the following tree model which uses only the training dataset:

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

Now we have a model built on approximately 70 % of the original dataset. You may remember before that we "cheated" in assessing the performance of our model by using observations that were already involved in the building of the model to test it. However, now we don't need to do this as we have approximately 30 % of the original dataset available for testing which have not been included in building the model.

3.2 Use the test set in predictions

The next step, once we have our model, is to generate predictions. We do this in the same way as in our previous tree labs, except this time using the test set to model the predictions. The (abridged) code for predictions is:

```
> predict(object, newdata = list(),type = c( "prob", "class", ))
```

where

- `object` is the object returned from the `rpart` command
- `newdata=list()` is the name of the dataset you wish to use. Usually this is just the test set.
- `type` defines whether you want probabilities of class membership. Default = probabilities which is what we want.

To implement this in our model:

```
> test$predprob<-predict(fit,test)
> summary(test$predprob)
```

This produces an object with two columns corresponding to the probabilities of class 1 and class 2. 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. In this case they are ordered alphabetically. (N before Y). To access the 1st column type `predprob[1,]`. For the second column substitute 2 for 1. If you use the type parameter and set it to `class` it will use the default cutoff of p=.5 and it contains just one column - predicted class.

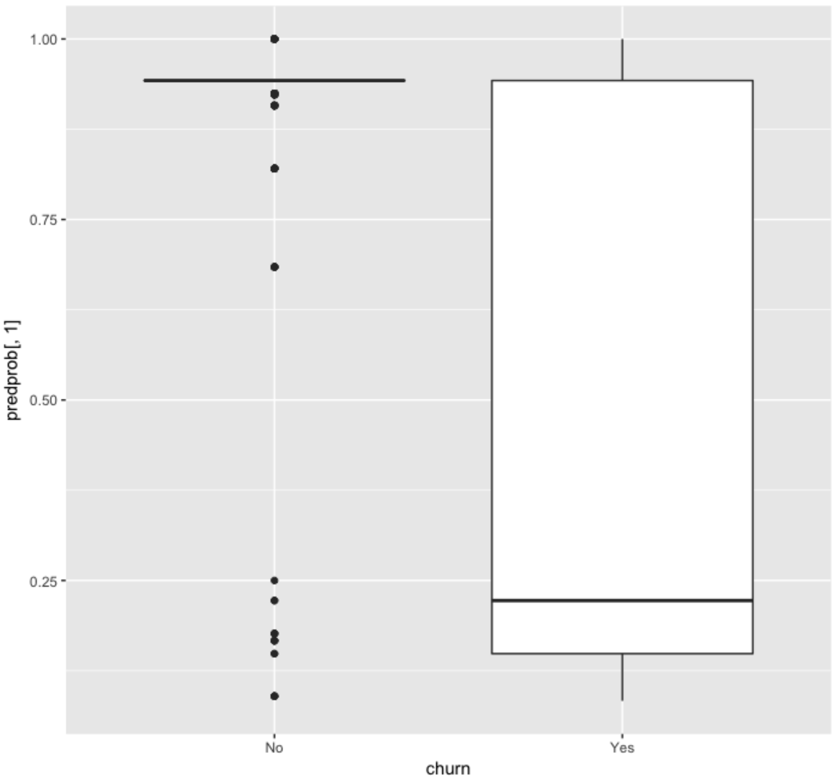
4 Boxplots and confusion tables

4.1 Boxplots

It is very easy to create a boxplot of predicted values vs target values. We already have our predicted probabilities stored under `predprob` so run the following:

```
> ggplot(data=test,aes(x=churn,y=predprob[,1]))+geom_boxplot()
```

You should get a plot similar to this....



Exercise: What is the alternative boxplot we could have constructed? Implement the changes required in the code above and examine the change in the plot. Hint: use column 2 of the predprob object.

5 ROC curves- the very basics

We are going to use the package `ROCR` to plot our data in various ways. This is an exceptionally useful library for carrying out model evaluations. You first of all have to create a set of predicted values using the usual command after the model you have fitted as above. In `ROCR` the first category is termed the negative category and the second the positive category. The second category or column should be taken for all the `ROCR` graphs. You can reverse this process if required. Consult the `ROCR` documentation for more information. The first thing is to use the `prediction`, a command contained within the `ROCR` package. Note that we are using the second column of the predicted values in this case. **It is very easy to forget this step**

```
> predics<-prediction(predprob[,2],test$churn)
```

Our `predics` object is what is known as an S4 object and we can see what is in it by running:

```
> str(predics)
```

You can also use the **attr** command.

You can see the object `predics` contains a number of slots(or variables) which can be used for future analyses. The slots in the object `predics` are accessed as follows:

```
predics@<variable name>
```

Note the use of the character '@'.

Exercise: Try the following commands and see if you can figure out what they are returning:

```
> predics@fp
> predics@fn
> predics@n.neg
```

For more on accessing the slots in an S4 object there is an excellent demo attached to `ROCR`. You can access it by typing `demo(ROCR)`. Study the R code to see what you can do.

6 Exploring `ROCR`

The package `ROCR` allows you calculate various performances measures for the x and y axes. We are going to use a very brief version today to plot the ROC curve and to calculate the area under the curve. Next week's lab will contain a lot more detail.

Assuming that you created the object `predics` as above, the following code will plot the ROC curve and calculate the Area Under the Curve

```
> perf<-performance(predics, "tpr","fpr")
> plot(perf)
>auc<-performance(predics,"auc")@y.values
>auc
```

7 Another package for ROC curves

There is another package called `pROC` which is worth while exploring. It produces confidence intervals for the area under the curve.

8 Confusion tables

8.1 Using the SDMTools package

There are many ways to create confusion tables in R. This section will show two ways. The first method uses a function called **confusion.matrix** included in the **SDMTools** library.

Before you use this package you have to recode outcome variable to 0 - negative event and 1 positive event. Again there are many ways to do in in **R**. The simplest(I think) is as follows:

```
test$nchurn<-ifelse(test$churn=="Yes",1,0)
```

Always check the results
You should also look at the **dplyr** and **car** package for other ways to recode data.

To create the confusion matrix for a specific threshold (in this 0.7) use

```
library(SDMTools)
confusion.matrix(test$nchurn,test$predprob[,2],threshold = 0.7)
```

Here we converted ”Yes” to 1 and hence used the second column of **predprob**

8.2 Writing a function to create confusion tables

This is a great example of how a function is written in `R`. We will need a function which takes:

1. our prediction probabilities for yes ($\mathbb{P}[\text{churn} = \text{Yes} \,|\, \mathbf{x}]$);
2. our chosen cut-off α; and which then returns actual Yes/No predictions. Below is a template with one omission.

The following is the code for a user defined `R` function. You can see that it uses an `if` statement nested within a `for` loop. The function name is `makePrediciton` and it takes two parameters as inputs - `probYes` and `cutoff`.

```
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.

With this new code we can now 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"))
```