Introduction to Classification

Advanced Data Mining

Dr. Simon Caton, and Dr. Cristian Rusu

Introduction

In this lab we are going to explore the fundamentals of classification:

- Sample a dataset (titanic in this case)
- Define the process to make a classification
- Classify data
- Build some simple models
- Compute a simple performance measure (accuracy)

Problem Definition: Surviving the Titanic

We are going to be working with the Titanic Dataset from Kaggle for a few weeks, so it makes sense to use it to explain the basics of classification. Note that R lab 3 will discuss additional information, and it is assumed that you have familiarised yourself with it for ADM week 2, as it goes through a lot of the data preparation steps we need for the next labs.

In titanic, our outcome measure (Y) is whether the individual survived or not. Our features (X) then determine the context for each observation of Y:

Attirbute	Description
PassengerId	an incremental ID
survival	(0 = No; 1 = Yes)
pclass	Passenger Class (1 = 1st; 2 = 2nd; 3 = 3rd)
name	Name
sex	Sex
age	Age
sibsp	Number of Siblings/Spouses Aboard
parch	Number of Parents/Children Aboard
ticket	Ticket Number
fare	Passenger Fare
cabin	Cabin
embarked	Port of Embarkation (C = Cherbourg;
^	Q = Queenstown; S = Southampton)

For the moment, we are only going to deal with the survival attribute.

Now switch your working directory to where you have downloaded the file on moodle. For me, that would be as follows:

```
setwd("/Users/simoncaton/Downloads/")
```

If you are a Windows user, remember to escape your \ s, as \ is a reserved character and so must be escaped. E.g.: *C:\\some directory\\some other directory*.

Now we are ready to read in our csv and put it in a data.frame (if you forgot / don't know what a data.frame is see R lab 2).

```
titanicData <- read.csv("titanic.csv", header=T, na.strings=c(""), stringsAsFactors = T)
```

We will come back to dealing with X later. For now let's just isolate Y.

```
y <- titanicData$Survived
```

We're left now with 891 observations (run: length(y) to see) of one variable:

```
table(y)
```

```
## y
##   0   1
## 549 342
```

We'll change the encoding of the variable to denote it as a categorical value, as opposed to an int (as it is now), and we'll be ready to run through the basics of classification.

```
y <- factor(y, levels = c(0,1), labels = c("No", "Yes"))
table(y)
```

```
## y
##  No Yes
## 549 342
```

Explore the data

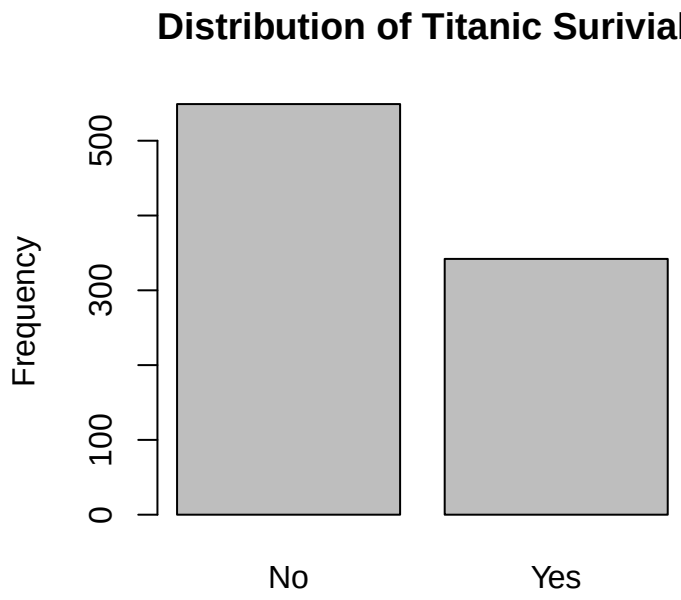
Based only on this one feature (Y), what can we tell about our data? Well, it's imbalanced: more people perished than survived. Or to be more specific:

```
prop.table(table(y))
```

```
## y
##        No       Yes
## 0.6161616 0.3838384
```

Or

```
barplot(table(y), main = "Distribution of Titanic Surivial", ylab="Frequency")
```



Whilst this imbalance is not as bad as it could be, it will distort simple notions of performance, like prediction accuracy: were we to always predict No, we could expect to be right about 61% of the time. As we move through the course, we'll encounter other forms of data exploration. For the moment, just make sure that data exploration (or exploratory data analysis) is always the first thing you do with any dataset. For a more thorough example of data exploration of the titantic dataset see R lab 3 (on moodle).

Building a simple model

Recall that we want to predict whether someone does / does not survive the titanic.

Let's for the moment, assume our predictive model is a coin: heads they survive, tails, they don't. Whilst this is a little macabre, it illustrates one of the simplest models we can build. For comparison let's also build a second model, that always predicts someone will not survive. You'll add a few more later.

Sampling the data

We need to operate on a portion (or sample) of our data. However, as we are not actually training a model yet, we just need some testing data. For simplicity, let's randomly draw 25% of the data.

```
set.seed(1337)
index <- sample(1:length(y), length(y) * .25, replace=FALSE)
testing <- y[index]
```

This randomly selects 25% of the rows, and then subsets y accordingly. The set.seed function just provides a seed value for R's pseudo-random number generator, the number you provide here is essentially irrelevant. Setting a seed in this way ensures the reproducibility of the random number generation.

Building our "model"

Our dependent variable takes either the value Yes or No, so all we need to represent our classification model is a function that produces either of these values according to the specification defined earlier (random coin toss, or never survive). In fact the process of *training* a classification model is essentially learning a function whose output is either Yes or No.

The everyone perishes model is easy enough. We just need to know how many *No* values to produce:

```
perishModel <- rep("No", length(testing))
```

For our coin toss model, we need to generate as many random numbers between 0 and 1 as instances we have to predict, and round them. Then transform 0/1 to No/Yes.

```
coinModel <- round(runif(length(testing), min=0, max=1))
coinModel <- factor(coinModel, levels = c(0,1), labels = c("No", "Yes"))
```

Strictly speaking, we should probably also transform our perish model:

```
perishModel <- factor(perishModel, levels = c("No", "Yes"), labels = c("No", "Yes"))
```

Now let's see how they do:

```
table(testing, perishModel)
```

```
##         perishModel
## testing  No Yes
##     No  136   0
##     Yes  86   0
```

To briefly explain what the table here shows:

vs.	Predicted: No	Predicted: Yes
Reality: No	Num Correct	Num Incorrect
Reality: Yes	Num Incorrext	Num Correct

What we see here is that our perish model always predicts the passenger will not surive (the left column). However, some passengers (instances) in our testing sample do survive, thus our table (actually called a coincidence matrix) illustrates the number of correct classifications (predicting No when No is correct), and incorrect classifications (predicting No when Yes is correct). As we will see in lecture 2, there are a lot of performance values that we can derive from this matrix.

Let's see how our coin toss does:

```
table(testing, coinModel)
```

```
##         coinModel
## testing No Yes
##     No  82  54
##     Yes 43  43
```

Determining Prediction Accuracy

For the sake of completeness, let's convert this into the % of correctly classified instances, also called prediction accuracy.

```
(coinAccuracy <- 1 - mean(coinModel != testing))
```

```
## [1] 0.5630631
```

```
(perishAccuracy <- 1 - mean(perishModel != testing))
```

```
## [1] 0.6126126
```

Taking the mean works here because *coinModel != testing* returns a vector of true/false values (essentially the pair wise equality test for all values in the coinModel and testing vectors). When we take the mean of a vector of 0s (false) and 1s (true) we get the percentage of 1s (true) in the vector, or in this case, the percentage accuracy of our model(s).

Recall that the class imbalance in the dataset is around 61%, so a model performance of 61 is not as good as it looks.

Adding robustness to our model evaluation

To test or models we randomly sampled our data. We know the original distribution of Y, but we didn't actually check the distribution of our sample.

```
prop.table(table(testing))
```

```
## testing
##        No       Yes
## 0.6126126 0.3873874
```

Not too surprisingly, it is the same as the model performance of the perishModel. However, we should probably try our models on more than one sample. Were we also training a model, the same would be true of any training dataset.

Let's get some observations of model performance, but making more models, on more samples of the data. We'll build and evaluate 1000 models on 1000 samples to get an idea of how each performs.

```
perish <- c()
coin <- c()

for (i in 1:1000) {
  index <- sample(1:length(y), length(y) * .25, replace=FALSE)
  testing <- y[index]

  coinModel <- round(runif(length(testing), min=0, max=1))
  coinModel <- factor(coinModel, levels = c(0,1), labels = c("No", "Yes"))

 coin[i] <- 1 - mean(coinModel != testing)
 perish[i] <- 1 - mean(perishModel != testing)
}

results <- data.frame(coin, perish)
names(results) <- c("Coin Toss Accuracy", "Everyone Perishes Accuracy")
summary(results)
```

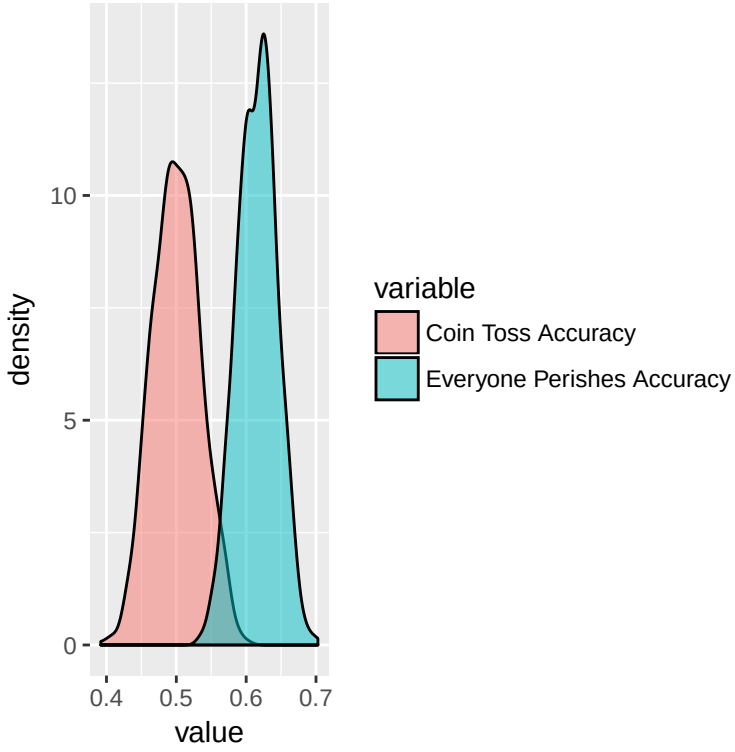
```
##  Coin Toss Accuracy Everyone Perishes Accuracy
##  Min.   :0.3919     Min.   :0.5315
##  1st Qu.:0.4775     1st Qu.:0.5946
##  Median :0.5000     Median :0.6171
##  Mean   :0.5005     Mean   :0.6155
##  3rd Qu.:0.5225     3rd Qu.:0.6351
##  Max.   :0.6036     Max.   :0.7027
```

At this point, we can compute any (appropriate) statistical test to compare the performance. These models ran quickly (the are essentially arbitrary), as we increase the level of sophistication, the runtime required will also increase!

If you want a visual comparison, we can use density plots to get an idea of the distributions.

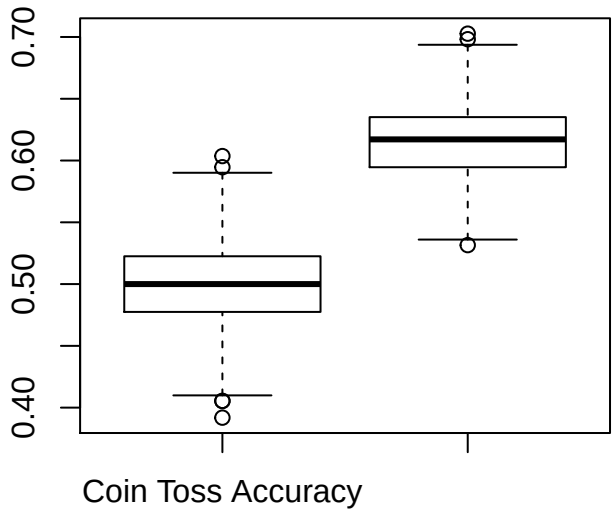
```
library(ggplot2)
library(reshape)
ggplot(melt(results), mapping = aes (fill = variable, x = value)) + geom_density (alpha = .5)
```

```
## Using  as id variables
```



Or the box plot.

```
boxplot(results)
```



In both cases, we can see that the everyone perishes model seems to perform better, this is essentially an artefact of the class imbalance; more people perished than survived, so always predicting perish is a *safe* albeit sombre choice.

Using Independent Variables

For the sake of completeness, let's use some of the dataset to illustrate how model training works.

It's a fairly safe assumption that gender plays a key role in whether someone survived or not, with the old adage of **women and children first**. Other fairly safe assumptions would be the class of travel, age, and

potentially having relatives on board. We'll make one last model to show how we can use training data gender in this case.

Let's extend our data.frame to include gender, and get explore the relationship between the the dependent and independent variables.

```
df <- titanicData[, c("Survived", "Sex")]
df$Survived <- factor(df$Survived, levels = c(0,1), labels = c("No", "Yes"))
```

To play fair, we need to split our data into testing and training subsets. Otherwise, we have complete information, which may bias our model building.

```
index <- sample(1:dim(df)[1], dim(df)[1] * .75, replace=FALSE)
training <- df[index, ]
testing <- df[-index, ]

table(training$Survived, training$Sex)
```

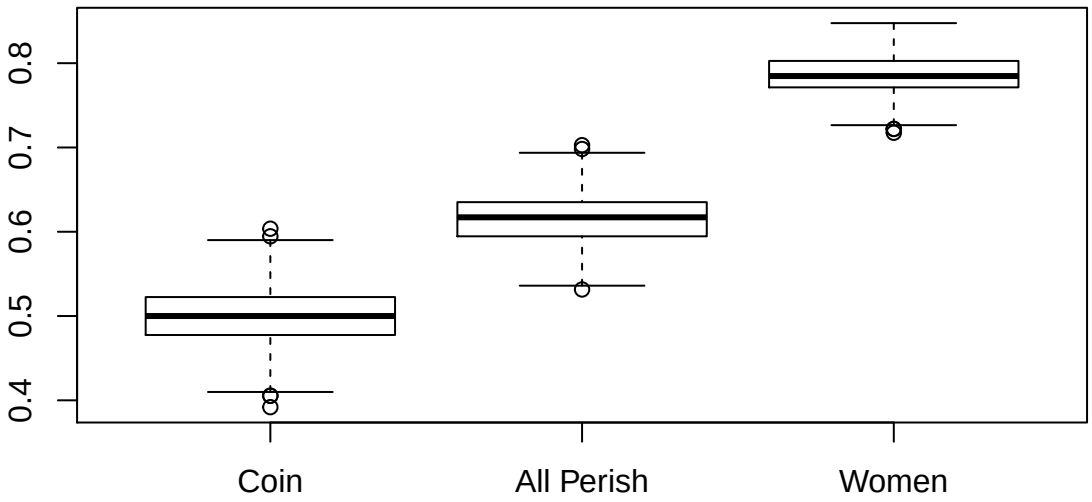
```
##
##       female male
##   No      59  348
##   Yes    178   83
```

Based on this very simple exercise we can see that gender may influence survival. Let's use this information to derive some rules:

- If female predict Yes
- If male predict No

Now, let us build this as a model (and declare a function for reuse later), and see how it does reusing (and slightly extending) our for loop from earlier.

```
predictSurvival <- function(data) {
  model <- rep("No", dim(data)[1])
  model[data$Sex == 'female'] <- "Yes"
  return(model)
}

women <- c()

for (i in 1:1000) {
  index <- sample(1:dim(df)[1], dim(df)[1] * .75, replace=FALSE)
  testing <- df[-index, ]

  womenModel <- predictSurvival(testing)

  women[i] <- 1 - mean(womenModel != testing$Survived)
}

results$`Women Accuracy` <- women
names(results) <- c("Coin", "All Perish", "Women")

boxplot(results)
```



This is essentially, what statistical learning and machine learning models do. They model the dependent variable Y using the independent variable(s) X.

Note also that the simplicity of this approach, should also be a pretty big indicator, that a prediction accuracy of less than 80% probably isn't very good for this data. However, taken out of this context, 80% sounds pretty good. If you take nothing else away from this lab, make sure it's to always question: how challenging it is to have achieved "high" accuracy?

Exercise

Add more rules, and see which does best:

1. Age (children vs. adults)
2. Passenger Class
3. Women + children etc.