Machine Learning with R

François de Ryckel

2017-11-19

Contents

List of Tables

List of Figures

Chapter 1

Prerequisites

Welcome to my reference book in machine learning. I have tried to put in it all the tricks, tips, how-to, must-know, etc. I consult it almost everytime I embark on data science project. It is impossible to remember all the coding practices, hence this my data science in R bible. This book is basically a record of my journey in data analysis. So often, I spend time reading articles, blog posts, etc. and wish I could put all the things I'm learning in a central location. It is a living document with constant additions.

So this book is a compilation of the techniques I've learned along the way. Most of what I have learned is through blog posts, stack overflow questions, etc. I am not taking any credit for all the great ideas, examples, graphs, etc. in this web book. I do take responsibility for all mistakes, typos, unclear explanations, poor labeling / presentation of graphs. If you find anything that require improvement, I would be grateful if you would let me know: f.deryckel@gmail.com or even better post an issue on github here.

I am assuming that you are already somehow familiar with:

- the math behind most algorithms. This is not a math book.

- the basics of how to use R. This is not a computer science book nor an introductory R book.

I wish you loads of fun in your data science journey, and I hope that this book can contribute positively to your experience.

1.1 Pre-requisite and conventions

As much as it makes sense, we will use the tidyverse and the conventions of tidy data throughout our journey.
Besides the hype surrounding the tidyverse, there is a couple reasons for us to stick with it:

- learning a language is hard on itself. If we can be proficient and creative with one, it will be much better. All the packages from the tidyverse, might not always be the best ones (more efficient, more elegant), but I'm happy to learn inside out one opinionated framework in order to be able to apply it effortlessly and creatively.

- Because many of the tidyverse packages do their background work in C++, they are usually pretty efficient in the way they work.

```r
library(tidyverse)
```

Here are some conventions we will be using throughout the book.

- `df` denotes a data frame. Usually the data frame from a raw set of data
- We'll use `df2`, `df3`, etc. for other, "cleaner" versions of that raw data set
- `model_pca_xxxx`, `model_lr_xxxx` denotes models. The second part denotes the algorithm.
- `predict_svm_xxxx` or `predict_mlr_xxxx` denotes the outcome of applying a model on a set of independent variables.

1.2 Organization

The first part of the book is more about the nitty-gritty of each machine learning algorithm. We do not really go into the depth of how they work and why they work the way they do. Instead it is really on how to leveraged R and various R libraries to use the ML algorithms. The second part of the book is about various case studies. Most of them come either from the UCI Machine learning repository or Kaggle.

The two parts can (and maybe should?) be read concommitenly. We use machine learning to model real-life situation, so I see it as essential to go from the algortihms and theory to the case study and practical applications.

So in the first part, we start by talking about inference and tests with the Chapter 2. We then go onto the various linear regression technique with the Chapter 3. Chapter 4 is about logisitic regression and the various way to evaluate a logisitic model. We then go onto the K Nearest Neighbour with 6.

The case studies have been put by order of skills required to approach the practical situation.

Chapter 2

Tests and inferences

Definitely the first thing to be familiar with while doing machine learning works is the basic of statistical inferences.
In this chapter, we go over some of these important concepts and the r-ways to do them.

Let's get started.

2.1 Assumption of normality

Copied from here

Many of the statistical procedures including correlation, regression, t tests, and analysis of variance, namely parametric tests, are based on the assumption that the data follows a normal distribution or a Gaussian distribution (after Johann Karl Gauss, 1777–1855); that is, it is assumed that the populations from which the samples are taken are normally distributed. The assumption of normality is especially critical when constructing reference intervals for variables. Normality and other assumptions should be taken seriously, for when these assumptions do not hold, it is impossible to draw accurate and reliable conclusions about reality.

With large enough sample sizes (> 30 or 40), the violation of the normality assumption should not cause major problems; this implies that we can use parametric procedures even when the data are not normally distributed (8). If we have samples consisting of hundreds of observations, we can ignore the distribution of the data (3). According to the central limit theorem,

- if the sample data are approximately normal then the sampling distribution too will be normal;
- in large samples (> 30 or 40), the sampling distribution tends to be normal, regardless of the shape of the data
- means of random samples from any distribution will themselves have normal distribution.

Although true normality is considered to be a myth, we can look for normality visually by using normal plots or by significance tests, that is, comparing the sample distribution to a normal one. It is important to ascertain whether data show a serious deviation from normality.

2.1.1 Visual check of normality

Visual inspection of the distribution may be used for assessing normality, although this approach is usually unreliable and does not guarantee that the distribution is normal. However, when data are presented visually, readers of an article can judge the distribution assumption by themselves. The frequency distribution (histogram), stem-and-leaf plot, boxplot, P-P plot (probability-probability plot), and Q-Q plot (quantile-quantile plot) are used for checking normality visually. The frequency distribution that plots the observed values against their frequency, provides both a visual judgment about whether the distribution is bell shaped and insights about gaps in the data and outliers outlying values. A Q-Q plot is very similar to the P-P plot except that it plots the quantiles (values that split a data set into equal portions) of the data set instead of every individual score in the data. Moreover, the Q-Q plots are easier to interpret in case of large sample sizes. The boxplot shows the median as a horizontal line inside the box and the interquartile range (range between the 25 th to 75 th percentiles) as the length of the box. The whiskers (line extending from the top and bottom of the box) represent the minimum and maximum values when they are within 1.5 times the interquartile range from either end of the box. Scores greater than 1.5 times the interquartile range are out of the boxplot and are considered as outliers, and those greater than 3 times the interquartile range are extreme outliers. A boxplot that is symmetric with the median line at approximately the center of the box and with symmetric whiskers that are slightly longer than the subsections of the center box suggests that the data may have come from a normal distribution.

2.1.2 Normality tests

The various normality tests compare the scores in the sample to a normally distributed set of scores with the same mean and standard deviation; the null hypothesis is that "sample distribution is normal." If the test is significant, the distribution is non-normal. For small sample sizes, normality tests have little power to reject the null hypothesis and therefore small samples most often pass normality tests. For large sample sizes, significant results would be derived even in the case of a small deviation from normality, although this small deviation will not affect the results of a parametric test. It has been reported that the K-S test has low power and it should not be seriously considered for testing normality (11). Moreover, it is not recommended when parameters are estimated from the data, regardless of sample size (12).

The Shapiro-Wilk test is based on the correlation between the data and the corresponding normal scores and provides better power than the K-S test even after the Lilliefors correction.

Power is the most frequent measure of the value of a test for normality. Some researchers recommend the Shapiro-Wilk test as the best choice for testing the normality of data.

2.2 T-tests

The **independent t test** is used to test if there is any statistically *significant difference between two means.* Use of an independent t test requires several assumptions to be satisfied.

1. The variables are continuous and independent
2. The variables are normally distributed
3. The variances in each group are equal

When these assumptions are satisfied the results of the t test are valid. Otherwise they are invalid and you need to use a non-parametric test. When data is not normally distributed you can apply transformations to make it normally distributed.

Using the `mtcars` data set, we check if there are any difference in mile per gallon (mpg) for each of the automatic and manual group.

Check the data and mark as factor the driving system.

```
library(tidyverse)
glimpse(mtcars)
```

```
## Observations: 32
## Variables: 11
## $ mpg  <dbl> 21.0, 21.0, 22.8, 21.4, 18.7, 18.1, 14.3, 24.4, 22.8, 19....
## $ cyl  <dbl> 6, 6, 4, 6, 8, 6, 8, 4, 4, 6, 6, 8, 8, 8, 8, 8, 8, 4, 4, ...
## $ disp <dbl> 160.0, 160.0, 108.0, 258.0, 360.0, 225.0, 360.0, 146.7, 1...
## $ hp   <dbl> 110, 110, 93, 110, 175, 105, 245, 62, 95, 123, 123, 180, ...
## $ drat <dbl> 3.90, 3.90, 3.85, 3.08, 3.15, 2.76, 3.21, 3.69, 3.92, 3.9...
## $ wt   <dbl> 2.620, 2.875, 2.320, 3.215, 3.440, 3.460, 3.570, 3.190, 3...
## $ qsec <dbl> 16.46, 17.02, 18.61, 19.44, 17.02, 20.22, 15.84, 20.00, 2...
## $ vs   <dbl> 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, ...
## $ am   <dbl> 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, ...
## $ gear <dbl> 4, 4, 4, 3, 3, 3, 3, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 4, 4, ...
## $ carb <dbl> 4, 4, 1, 1, 2, 1, 4, 2, 2, 4, 4, 3, 3, 3, 4, 4, 4, 1, 2, ...
```

```
df <- mtcars
df$am <- factor(df$am, labels = c("automatic", "manual"))
df2 <- df %>% select(mpg, am)
glimpse(df2)
```

```
## Observations: 32
## Variables: 2
## $ mpg <dbl> 21.0, 21.0, 22.8, 21.4, 18.7, 18.1, 14.3, 24.4, 22.8, 19.2...
## $ am  <fctr> manual, manual, manual, automatic, automatic, automatic, ...
```

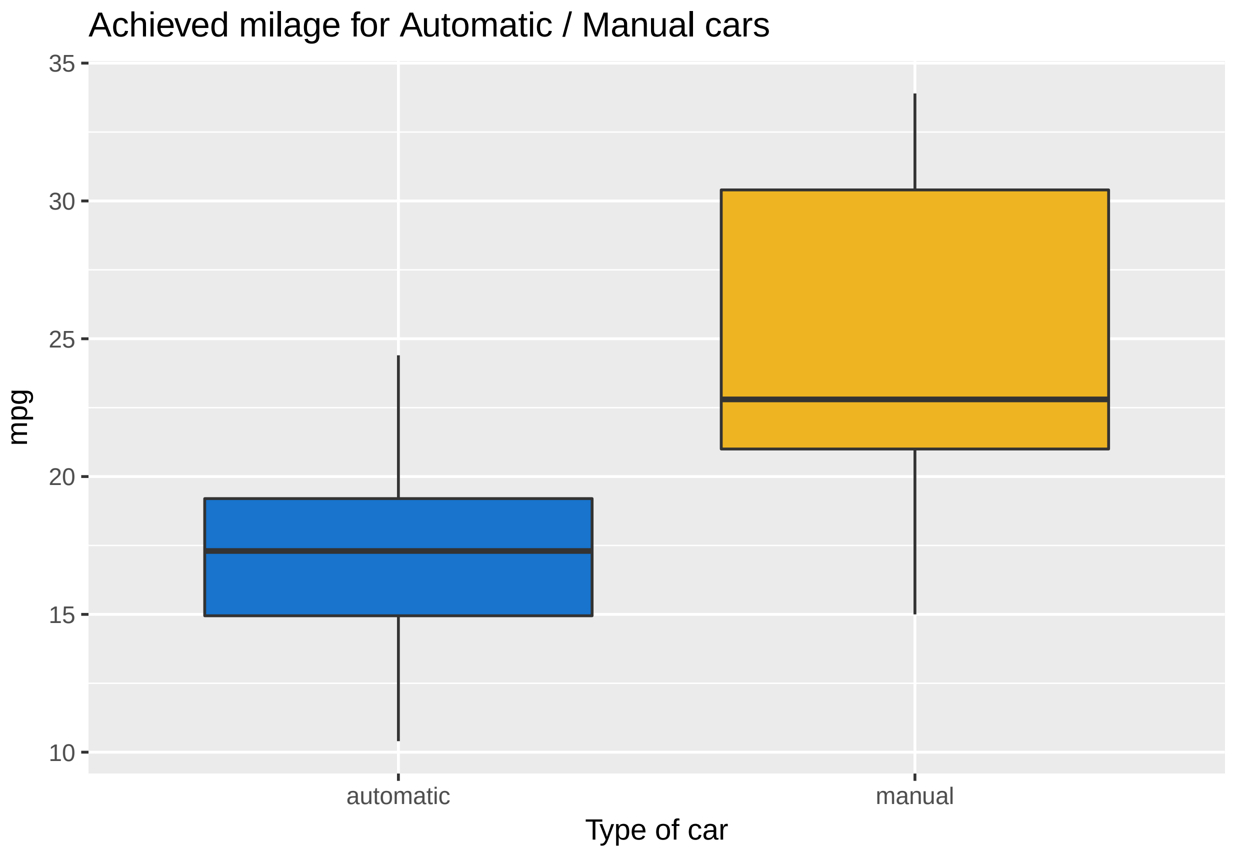
Generate descriptive statistic for each group.

```
df2 %>% group_by(am) %>%
  summarise(mean = mean(mpg), minimum = min(mpg), maximum = max(mpg))
```

```
## # A tibble: 2 x 4
##          am     mean minimum maximum
##      <fctr>    <dbl>   <dbl>   <dbl>
## 1 automatic 17.14737    10.4    24.4
## 2    manual 24.39231    15.0    33.9
```

Generate boxplot for each group

```
ggplot(df2, aes(x = am, y = mpg)) +
  geom_boxplot(fill = c("dodgerblue3", "goldenrod2")) +
  labs(title = "Achieved milage for Automatic / Manual cars",
       x = "Type of car")
```



Test the normality of the data.
To do so, we can use the **Shapiro Wilk Normality Test**

```
df2 %>% group_by(am) %>%
  summarise(shaprio_test = shapiro.test(mpg)$p.value)
```

```
## # A tibble: 2 x 2
##          am shaprio_test
##      <fctr>        <dbl>
## 1 automatic    0.8987358
## 2    manual    0.5362729
```

There is no evidence of departure from normality.

Test the equal variance in each group.
To do so, we use the `levene.test` from the `car` package.

```
car::leveneTest(mpg ~ am, center = mean, data = df2)
```

```
## Levene's Test for Homogeneity of Variance (center = mean)
##       Df F value  Pr(>F)
## group  1   5.921 0.02113 *
##       30
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Because the variance in the 2 groups is not equal, we have to transform the data.

Apply a log transformation to stabilize the variance.

```
log_transformed_mpg = log(df2$mpg)
```

Now we can finally apply the t test to our data.

```
t.test(log_transformed_mpg ~ df2$am, var.equal = TRUE)
```

```
##
##  Two Sample t-test
##
## data:  log_transformed_mpg by df2$am
## t = -3.9087, df = 30, p-value = 0.0004905
## alternative hypothesis: true difference in means is not equal to 0
## 95 percent confidence interval:
##  -0.5277597 -0.1655209
## sample estimates:
## mean in group automatic    mean in group manual
##                2.816692                3.163332
```

Interpret the results.

- Manual cars have on average a higher mileage per Gallon (24 mpg) compared to Automatic (17 mpg).

- The box plot did not reveal the presence of outliers
- The Shapiro-Wilk normality test did not show any deviation from normality in the data

- The Levene Test showed difference in the variance in the 2 group. We addressed that by log transforming the data

- The t test show a significant difference in the mean of miles per gallon from automatic and manual cars.

Chapter 3

Multiple Linear Regression

3.1 Single variable regression

The general equation for a linear regression model

$y^i = \beta_0 + \beta_1 x^i + \epsilon^i$

where:

- y^i is the i^{th} observation of the dependent variable
- β_0 is the intercept coefficient
- β_1 is the regression coefficient for the dependent variable
- x^i is the i^{th} observation of the independent variable
- ϵ^i is the error term for the i^{th} observation. It basically is the difference in therm of y between the observed value and the estimated value. It is also called the residuals. A good model minimize these errors. [1]

Some ways to assess how good our model is to:

1. compute the SSE (the sum of squared error)
   - SSE $= (\epsilon^1)^2 + (\epsilon^2)^2 + \ldots + (\epsilon^n)^2 = \sum_{i=1}^{N} \epsilon^i$
   - A good model will minimize SSE
   - problem: SSE is dependent of N. SSE will naturally increase as N increase
2. compute the RMSE (the root mean squared error)
   - RMSE $= \sqrt{\frac{SSE}{N}}$
   - Also a good model will minimize SSE
   - It depends of the unit of the dependent variable. It is like the average error the model is making (in term of the unit of the dependent variable)
3. compute R^2
   - It compare the models to a baseline model

[1] Remember that the error term, ϵ^i, in the simple linear regression model is independent of x, and is normally distributed, with zero mean and constant variance.

- R^2 is **unitless** and **universaly** interpretable
- SST is the sum of the squared of the difference between the observed value and the mean of all the observed value
- $R^2 = 1 - \frac{SSE}{SST}$

3.1.1 First example. Predicting wine price

The wine.csv file is used.

Let's load it and then have a quick look at its structure.

```
library(tidyverse)
library(skimr)
df = read_csv("dataset/Wine.csv")
skim(df)
```

```
## Skim summary statistics
##  n obs: 25
##  n variables: 7
##
## Variable type: integer
##           var missing complete  n    mean     sd  min  p25 median  p75
## 1         Age       0       25 25   17.2    7.69    5   11     17   23
## 2 HarvestRain       0       25 25  148.56  74.42   38   89    130  187
## 3  WinterRain       0       25 25  605.28 132.28  376  536    600  697
## 4        Year       0       25 25 1965.8    7.69 1952 1960   1966 1972
##    max     hist
## 1   31
## 2  292
## 3  830
## 4 1978
##
## Variable type: numeric
##         var missing complete  n     mean      sd      min      p25
## 1      AGST       0       25 25    16.51    0.68    14.98    16.2
## 2 FrancePop       0       25 25 49694.44 3665.27 43183.57 46584
## 3     Price       0       25 25     7.07    0.65     6.2      6.52
##     median      p75      max     hist
## 1    16.53    17.07    17.65
## 2 50254.97 52894.18 54602.19
## 3     7.12     7.5      8.49
```

We use the `lm` function to find our linear regression model. We use *AGST* as the independent variable while the *price* is the dependent variable.

```r
model_lm_df = lm(Price ~ AGST, data = df)
summary(model_lm_df)
```

```
##
## Call:
## lm(formula = Price ~ AGST, data = df)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.78450 -0.23882 -0.03727  0.38992  0.90318
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -3.4178     2.4935  -1.371 0.183710
## AGST          0.6351     0.1509   4.208 0.000335 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.4993 on 23 degrees of freedom
## Multiple R-squared:  0.435,  Adjusted R-squared:  0.4105
## F-statistic: 17.71 on 1 and 23 DF,  p-value: 0.000335
```

The `summary` function applied on the model is giving us a bunch of important information

- the stars next to the predictor variable indicated how significant the variable is for our regression model
- it also gives us the value of the R coefficient

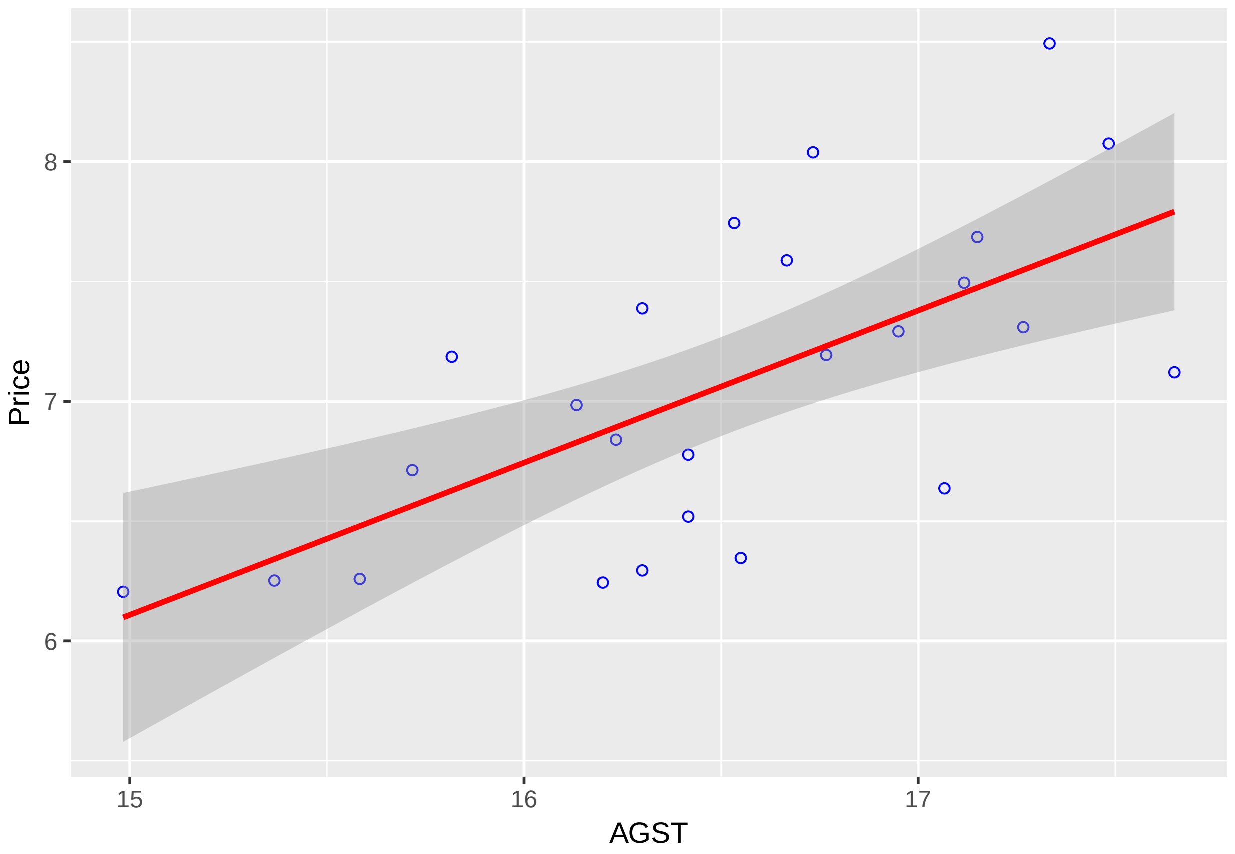
We could have calculated the R value ourselves:

```r
SSE = sum(model_lm_df$residuals^2)
SST = sum((df$Price - mean(df$Price))^2)
r_squared = 1 - SSE/SST
r_squared
```

```
## [1] 0.4350232
```

We can now plot the observations and the line of regression; and see how the linear model fits the data.

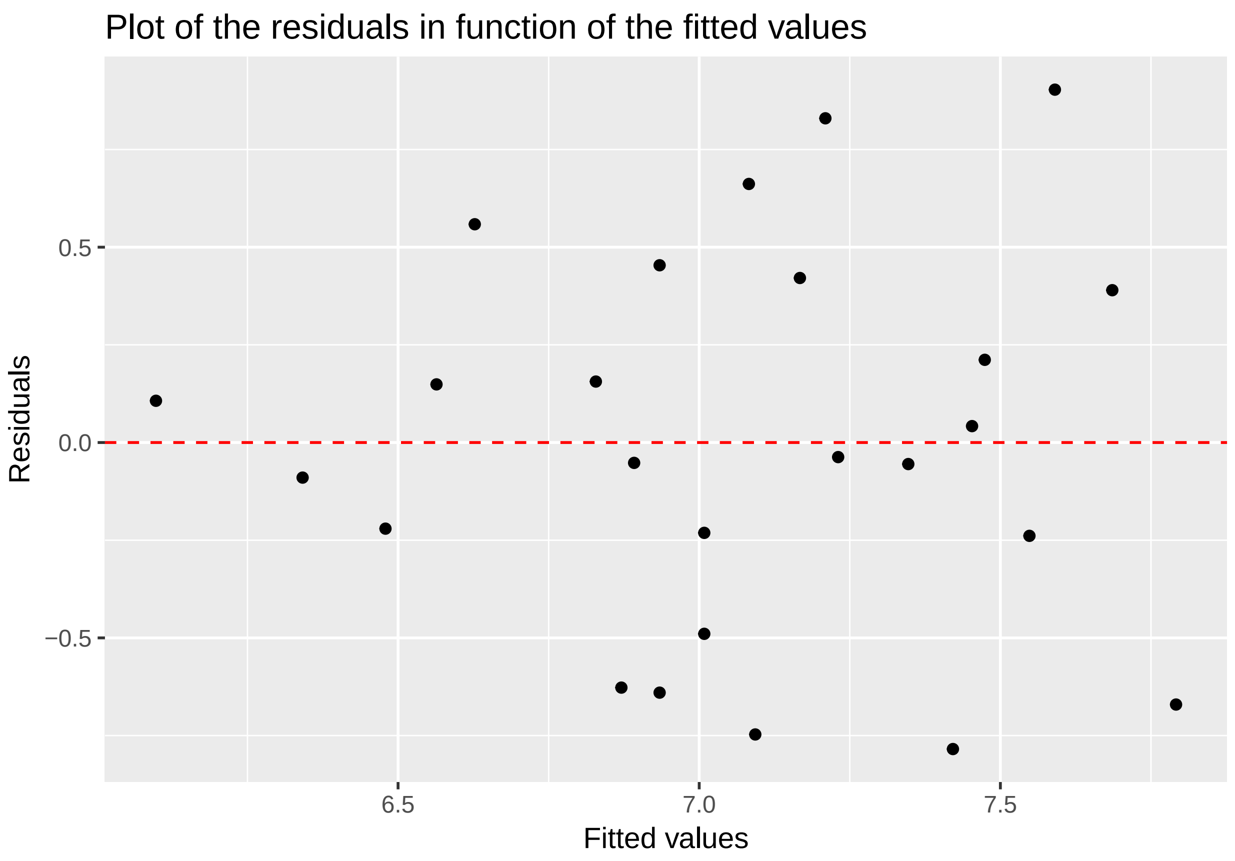
```r
ggplot(df, aes(AGST, Price)) +
  geom_point(shape = 1, col = "blue") +
  geom_smooth(method = "lm", col = "red")
```



By default, the `geom_smooth()` will use a 95% confidence interval (which is the grey-er area on the graph). There are 95% chance the line of regression will be within that zone for the whole population.

It is always nice to see how our residuals are distributed.
We use the `ggplot2` library and the `fortify` function which transform the `summary(model1)` into a data frame usable for plotting.

```
model1 <- fortify(model_lm_df)
p <- ggplot(model1, aes(.fitted, .resid)) + geom_point()
p <- p + geom_hline(yintercept = 0, col = "red", linetype = "dashed")
p <- p + xlab("Fitted values") + ylab("Residuals") + ggtitle("Plot of the residuals in 
p
```



3.2 Multi-variables regression

Instead of just considering one variable as predictor, we'll add a few more variables to our model with the idea to increase its predictive ability.

We have to be cautious in adding more variables. Too many variable might give a high R^2 on our training data, but this not be the case as we switch to our testing data.

The general equations can be expressed as

$$y^i = \beta_0 + \beta_1 x_1^i + \beta_2 x_2^i + \ldots + \beta_k x_k^i + \epsilon^i$$

when there are k predictors variables.

There are a bit of trials and errors to make while trying to fit multiple variables into a model, but a rule of thumb would be to include most of the variable (all these that would make sense) and then take out the ones that are not very significant using the `summary(modelx)`

3.2.1 First example. Predicting wine price

We continue here with the same dataset, *wine.csv*.
First, we can see how each variable is correlated with each other ones, using

```
cor(df)
```

```
##                    Year      Price   WinterRain        AGST HarvestRain
## Year         1.00000000 -0.4477679  0.016970024 -0.24691585  0.02800907
## Price       -0.44776786  1.0000000  0.136650547  0.65956286 -0.56332190
## WinterRain   0.01697002  0.1366505  1.000000000 -0.32109061 -0.27544085
## AGST        -0.24691585  0.6595629 -0.321090611  1.00000000 -0.06449593
## HarvestRain  0.02800907 -0.5633219 -0.275440854 -0.06449593  1.00000000
## Age         -1.00000000  0.4477679 -0.016970024  0.24691585 -0.02800907
## FrancePop    0.99448510 -0.4668616 -0.001621627 -0.25916227  0.04126439
##                     Age    FrancePop
## Year        -1.00000000  0.994485097
## Price        0.44776786 -0.466861641
## WinterRain  -0.01697002 -0.001621627
## AGST         0.24691585 -0.259162274
## HarvestRain -0.02800907  0.041264394
## Age          1.00000000 -0.994485097
## FrancePop   -0.99448510  1.000000000
```

by default, R uses the Pearson coefficient of correlation.
So let's start by using all variables.

```
model2_lm_df <- lm(Price ~ Year + WinterRain + AGST + HarvestRain + Age + FrancePop, dat
summary(model2_lm_df)
```

```
##
## Call:
## lm(formula = Price ~ Year + WinterRain + AGST + HarvestRain +
##     Age + FrancePop, data = df)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.48179 -0.24662 -0.00726  0.22012  0.51987
##
## Coefficients: (1 not defined because of singularities)
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)  7.092e-01  1.467e+02   0.005 0.996194
## Year        -5.847e-04  7.900e-02  -0.007 0.994172
## WinterRain   1.043e-03  5.310e-04   1.963 0.064416 .
## AGST         6.012e-01  1.030e-01   5.836 1.27e-05 ***
## HarvestRain -3.958e-03  8.751e-04  -4.523 0.000233 ***
## Age                 NA         NA      NA       NA
## FrancePop   -4.953e-05  1.667e-04  -0.297 0.769578
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
```

```
## Residual standard error: 0.3019 on 19 degrees of freedom
## Multiple R-squared:  0.8294, Adjusted R-squared:  0.7845
## F-statistic: 18.47 on 5 and 19 DF,  p-value: 1.044e-06
```

While doing so, we notice that the variable *Age* has NA (issues with missing data?) and that the variable *FrancePop* isn't very predictive of the price of wine. So we can refine our models, by taking out these 2 variables, and as we'll see, it won't affect much our R^2 value. Note that with multiple variables regression, it is important to look at the **Adjusted R-squared** as it take into consideration the amount of variables in the model.

```
model3_lm_df <- lm(Price ~ Year + WinterRain + AGST + HarvestRain, data = df)
summary(model3_lm_df)
```

```
##
## Call:
## lm(formula = Price ~ Year + WinterRain + AGST + HarvestRain,
##     data = df)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.45470 -0.24273  0.00752  0.19773  0.53637
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) 44.0248601 16.4434570   2.677 0.014477 *
## Year        -0.0239308  0.0080969  -2.956 0.007819 **
## WinterRain   0.0010755  0.0005073   2.120 0.046694 *
## AGST         0.6072093  0.0987022   6.152  5.2e-06 ***
## HarvestRain -0.0039715  0.0008538  -4.652 0.000154 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.295 on 20 degrees of freedom
## Multiple R-squared:  0.8286, Adjusted R-squared:  0.7943
## F-statistic: 24.17 on 4 and 20 DF,  p-value: 2.036e-07
```

Although it isn't now feasible to graph in 2D the *Price* in function of the other variables, we can still graph our residuals.

```
model3 <- fortify(model3_lm_df)
p <- ggplot(model3, aes(.fitted, .resid)) + geom_point()
p <- p + geom_hline(yintercept = 0, col = "red", linetype = "dashed") + xlab("Fitted val
p <- p + ylab("Residuals") + ggtitle("Plot of the residuals in function of the fitted va
```

Chapter 4

Logistic Regression

4.1 Introduction

Logistic Regression is a classification algorithm. It is used to predict a binary outcome (1 / 0, Yes / No, True / False) given a set of independent variables. To represent binary / categorical outcome, we use dummy variables. You can also think of logistic regression as a special case of linear regression when the outcome variable is categorical, where we are using log of odds as dependent variable. In simple words, it predicts the probability of occurrence of an event by fitting data to a logit function.
Logistic Regression is part of a larger class of algorithms known as Generalized Linear Model (glm).
Although most logisitc regression should be called **binomial logistic regression**, since the variable to predict is binary, however, logistic regression can also be used to predict a dependent variable which can assume more than 2 values. In this second case we call the model **multinomial logistic regression**. A typical example for instance, would be classifying films between "Entertaining", "borderline" or "boring".

4.2 The logistic equation.

The general equation of the **logit model**

$$\mathbf{Y} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n$$

where **Y** is the variable to predict.
β is the coefficients of the predictors and the x_i are the predictors (aka independent variables).
In logistic regression, we are only concerned about the probability of outcome dependent variable (success or failure). We should then rewrite our function

$$p = e^{(\beta_0+\beta_1 x_1+\beta_2 x_2+...+\beta_n x_n)}$$

.
This however does not garantee to have p between 0 and 1.
Let's then have

$$p = \frac{e^{(\beta_0+\beta_1 x_1+\beta_2 x_2+...+\beta_n x_n)}}{e^{(\beta_0+\beta_1 x_1+\beta_2 x_2+...+\beta_n x_n)}+1}$$

or

$$p = \frac{e^Y}{e^Y+1}$$

where p is the probability of success. With little further manipulations, we have

$$\frac{p}{1-p} = e^Y$$

and

$$\log\frac{p}{1-p} = Y$$

If we remember what was **Y**, we get

$$\log\frac{p}{1-p} = \beta_0+\beta_1 x_1+\beta_2 x_2+...+\beta_n x_n$$

This is the equation used in Logistic Regression. Here (p/1-p) is the odd ratio. Whenever the log of odd ratio is found to be positive, the probability of success is always more than 50%.

4.3 Performance of Logistic Regression Model

To evaluate the performance of a logistic regression model, we can consider a few metrics.

- **AIC (Akaike Information Criteria)** The analogous metric of adjusted R-squared in logistic regression is AIC. AIC is the measure of fit which penalizes model for the number of model coefficients. Therefore, we always prefer model with minimum AIC value.

- **Null Deviance and Residual Deviance** Null Deviance indicates the response predicted by a model with nothing but an intercept. Lower the value, better the model. Residual deviance indicates the response predicted by a model on adding independent variables. Lower the value, better the model.

- **Confusion Matrix** It is nothing but a tabular representation of Actual vs Predicted values. This helps us to find the accuracy of the model and avoid overfitting.

- We can calcualate the accuracy of our model by

$$\frac{TruePositives + TrueNegatives}{TruePositives + TrueNegatives + FalsePositives + FalseNegatives}$$

- From confusion matrix, **Specificity** and **Sensitivity** can be derived as

$$Specificity = \frac{TrueNegatives}{TrueNegative + FalsePositive}$$

  and

$$Sensitivity = \frac{TruePositive}{TruePositive + FalseNegative}$$

- **ROC Curve** Receiver Operating Characteristic(ROC) summarizes the model's performance by evaluating the trade offs between true positive rate (sensitivity) and false positive rate(1- specificity). For plotting ROC, it is advisable to assume p $>$ 0.5 since we are more concerned about success rate. ROC summarizes the predictive power for all possible values of p $>$ 0.5. The area under curve (AUC), referred to as index of accuracy(A) or concordance index, is a perfect performance metric for ROC curve. Higher the area under curve, better the prediction power of the model. The ROC of a perfect predictive model has TP equals 1 and FP equals 0. This curve will touch the top left corner of the graph.

4.4 Setting up

As usual we will use the `tidyverse` and `caret` package

```r
library(caret)      # For confusion matrix
library(ROCR)       # For the ROC curve
library(tidyverse)
```

We can now get straight to business and see how to model logisitc regression with R and then have the more interesting discussion on its performance.

4.5 Example 1 - Graduate Admission

We use a dataset about factors influencing graduate admission that can be downloaded from the UCLA Institute for Digital Research and Education

The dataset has 4 variables

- `admit` is the response variable

- `gre` is the result of a standardized test

- `gpa` is the result of the student GPA (school reported)
- `rank` is the type of university the student apply for (4 = Ivy League, 1 = lower level entry U.)

Let's have a quick look at the data and their summary. The goal is to get familiar with the data, type of predictors (continuous, discrete, categorical, etc.)

```
df <- read_csv("dataset/grad_admission.csv")
glimpse(df)
```

```
## Observations: 400
## Variables: 4
## $ admit <int> 0, 1, 1, 1, 0, 1, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0,...
## $ gre   <int> 380, 660, 800, 640, 520, 760, 560, 400, 540, 700, 800, 4...
## $ gpa   <dbl> 3.61, 3.67, 4.00, 3.19, 2.93, 3.00, 2.98, 3.08, 3.39, 3....
## $ rank  <int> 3, 3, 1, 4, 4, 2, 1, 2, 3, 2, 4, 1, 1, 2, 1, 3, 4, 3, 2,...
```

```
#Quick check to see if our response variable is balanced-ish
table(df$admit)
```

```
##
##   0   1
## 273 127
```

Well that's not a very balanced response variable, although it is not hugely unbalanced either as it can be the cases sometimes in medical research.

```
## Two-way contingency table of categorical outcome and predictors
round(prop.table(table(df$admit, df$rank), 2), 2)
```

```
##
##          1    2    3    4
##   0 0.46 0.64 0.77 0.82
##   1 0.54 0.36 0.23 0.18
```

It seems about right ... most students applying to Ivy Leagues graduate programs are not being admitted.

Before we can run our model, let's transform the `rank` explanatory variable to a factor.

```
df2 <- df
df2$rank <- factor(df2$rank)

# Run the model
```

```
model_lgr_df2 <- glm(admit ~ ., data = df2, family = "binomial")
summary(model_lgr_df2)
```

```
##
## Call:
## glm(formula = admit ~ ., family = "binomial", data = df2)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.6268  -0.8662  -0.6388   1.1490   2.0790
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -3.989979   1.139951  -3.500 0.000465 ***
## gre          0.002264   0.001094   2.070 0.038465 *
## gpa          0.804038   0.331819   2.423 0.015388 *
## rank2       -0.675443   0.316490  -2.134 0.032829 *
## rank3       -1.340204   0.345306  -3.881 0.000104 ***
## rank4       -1.551464   0.417832  -3.713 0.000205 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 499.98  on 399  degrees of freedom
## Residual deviance: 458.52  on 394  degrees of freedom
## AIC: 470.52
##
## Number of Fisher Scoring iterations: 4
```

The next part of the output shows the coefficients, their standard errors, the z-statistic (sometimes called a Wald z-statistic), and the associated p-values. Both gre and gpa are statistically significant, as are the three terms for rank. The logistic regression coefficients give the change in the log odds of the outcome for a one unit increase in the predictor variable.
For every one unit change in `gre`, the log odds of admission (versus non-admission) increases by 0.002.
For a one unit increase in `gpa`, the log odds of being admitted to graduate school increases by 0.804.
The indicator variables for `rank` have a slightly different interpretation. For example, having attended an undergraduate institution with rank of 2, versus an institution with a rank of 1, changes the log odds of admission by -0.675.

To see how the variables in the model participates in the decrease of *Residual Deviance*, we can use the `ANOVA` function on our model.

```r
anova(model_lgr_df2)
```

```
## Analysis of Deviance Table
##
## Model: binomial, link: logit
##
## Response: admit
##
## Terms added sequentially (first to last)
##
##
##      Df Deviance Resid. Df Resid. Dev
## NULL                   399     499.98
## gre   1  13.9204       398     486.06
## gpa   1   5.7122       397     480.34
## rank  3  21.8265       394     458.52
```

We can test for an overall effect of `rank` (its significance) using the `wald.test function` of the `aod` library. The order in which the coefficients are given in the table of coefficients is the same as the order of the terms in the model. This is important because the wald.test function refers to the coefficients by their order in the model. We use the wald.test function. `b` supplies the coefficients, while `Sigma` supplies the variance covariance matrix of the error terms, finally `Terms` tells R which terms in the model are to be tested, in this case, terms 4, 5, and 6, are the three terms for the levels of `rank`.

```r
library(aod)
wald.test(Sigma = vcov(model_lgr_df2), b = coef(model_lgr_df2), Terms = 4:6)
```

```
## Wald test:
## ----------
##
## Chi-squared test:
## X2 = 20.9, df = 3, P(> X2) = 0.00011
```

The chi-squared test statistic of 20.9, with three degrees of freedom is associated with a p-value of 0.00011 indicating that the overall effect of rank is statistically significant.

Let's check how our model is performing. As mentioned earlier, we need to make a choice on the cutoff value (returned probability) to check our accuracy. In this first example, let's just stick with the usual `0.5` cutoff value.

```r
prediction_lgr_df2 <- predict(model_lgr_df2, data = df2, type = "response")
head(prediction_lgr_df2, 10)
```

```
##         1         2         3         4         5         6         7
## 0.1726265 0.2921750 0.7384082 0.1783846 0.1183539 0.3699699 0.4192462
##         8         9        10
## 0.2170033 0.2007352 0.5178682
```

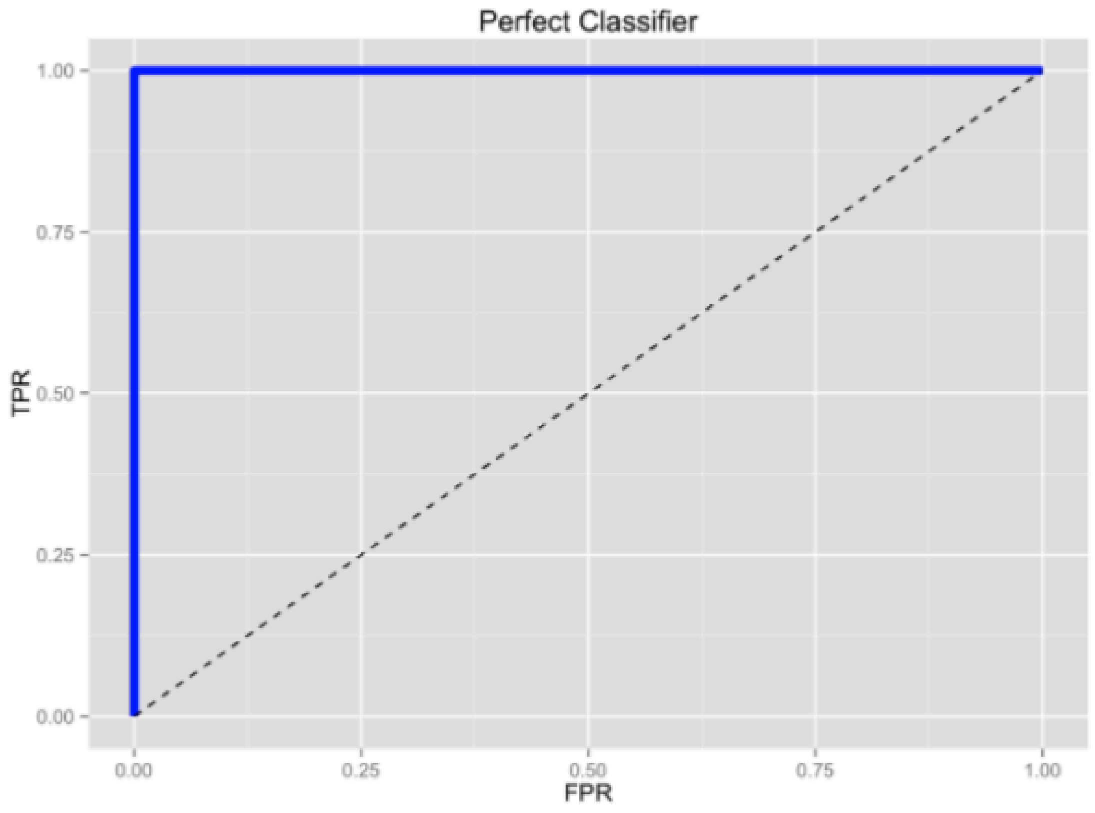
As it stands, the `predict` function gives us the probabilty that the observation has a response of 1; in our case, the probability that a student is being admitted into the graduate program. To check the accuracy of the model, we need a confusion matrix with a cut off value. So let's clean that vector of probability.

```
prediction_lgr_df2 <- if_else(prediction_lgr_df2 > 0.5 , 1, 0)
confusionMatrix(data = prediction_lgr_df2,
                reference = df2$admit, positive = "1")
```

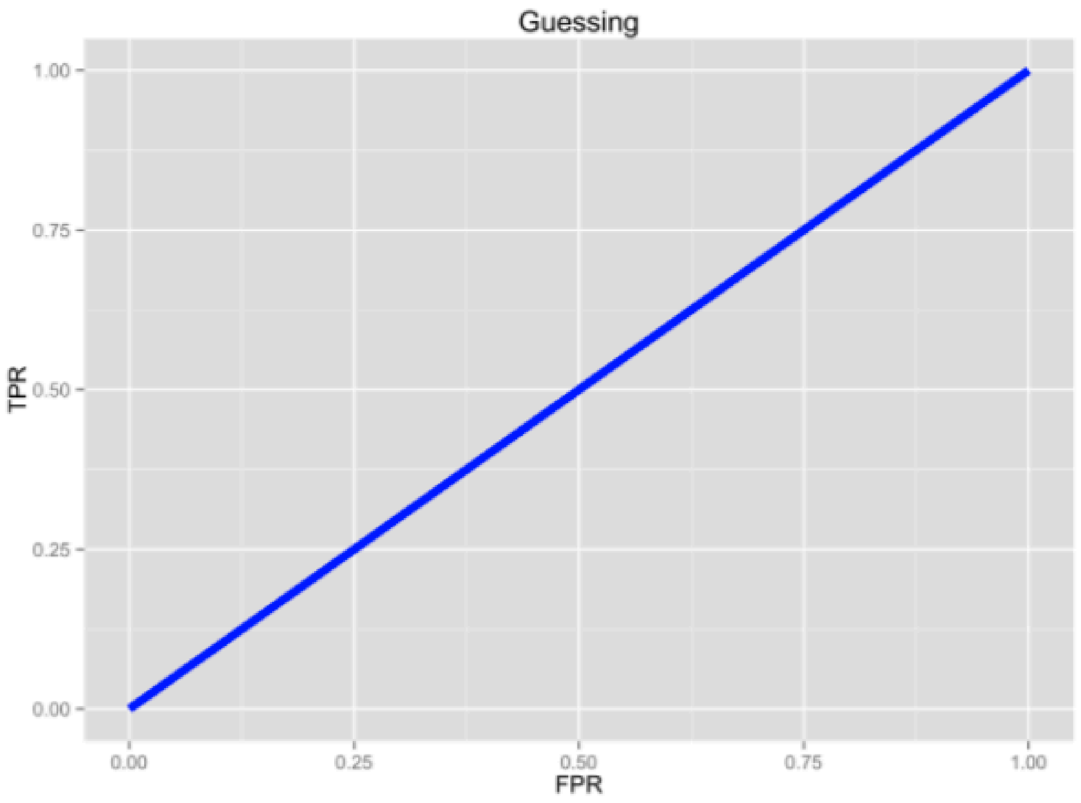
```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 254  97
##          1  19  30
##
##                Accuracy : 0.71
##                  95% CI : (0.6628, 0.754)
##     No Information Rate : 0.6825
##     P-Value [Acc > NIR] : 0.1293
##
##                   Kappa : 0.1994
##  Mcnemar's Test P-Value : 8.724e-13
##
##             Sensitivity : 0.2362
##             Specificity : 0.9304
##          Pos Pred Value : 0.6122
##          Neg Pred Value : 0.7236
##              Prevalence : 0.3175
##          Detection Rate : 0.0750
##    Detection Prevalence : 0.1225
##       Balanced Accuracy : 0.5833
##
##        'Positive' Class : 1
##
```

We have an interesting situation here. Although all our variables were significant in our model, the accuracy of our model, `71%` is just a little bit higher than the basic benchmark which is the no-information model (ie. we just predict the highest class) in this case `68.25%`.

Before we do a ROC curve, let's have a quick reminder on ROC.
ROC are plotting the proprotion of TP to FP. So ideally we want to have 100% TP and 0% FP.

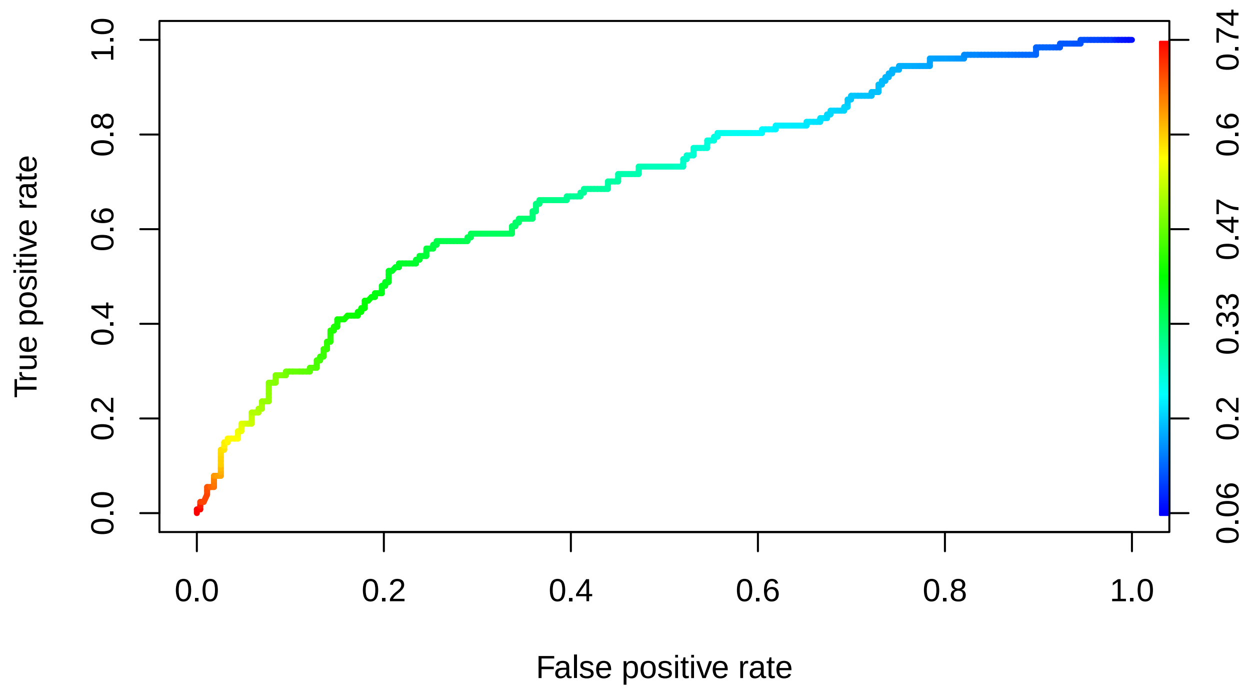


Pure Random guessing should lead to this curve



With that in mind, let's do a ROC curve on out model

```
prediction_lgr_df2 <- predict(model_lgr_df2, data = df2, type="response")
pr_admission <- prediction(prediction_lgr_df2, df2$admit)
prf_admission <- performance(pr_admission, measure = "tpr", x.measure = "fpr")
plot(prf_admission, colorize = TRUE, lwd=3)
```



At least it is better than just random guessing.

In some applications of ROC curves, you want the point closest to the TPR of 1 and FPR of 0. This cut point is “optimal” in the sense it weighs both sensitivity and specificity equally. Now, there is a cost measure in the ROCR package that you can use to create a performance object. Use it to find the cutoff with minimum cost.

```
cost_admission_perf = performance(pr_admission, "cost")
cutoff <- pr_admission@cutoffs[[1]][which.min(cost_admission_perf@y.values[[1]])]
```

Using that cutoff value we should get our sensitivity and specificity a bit more in balance. Let’s try

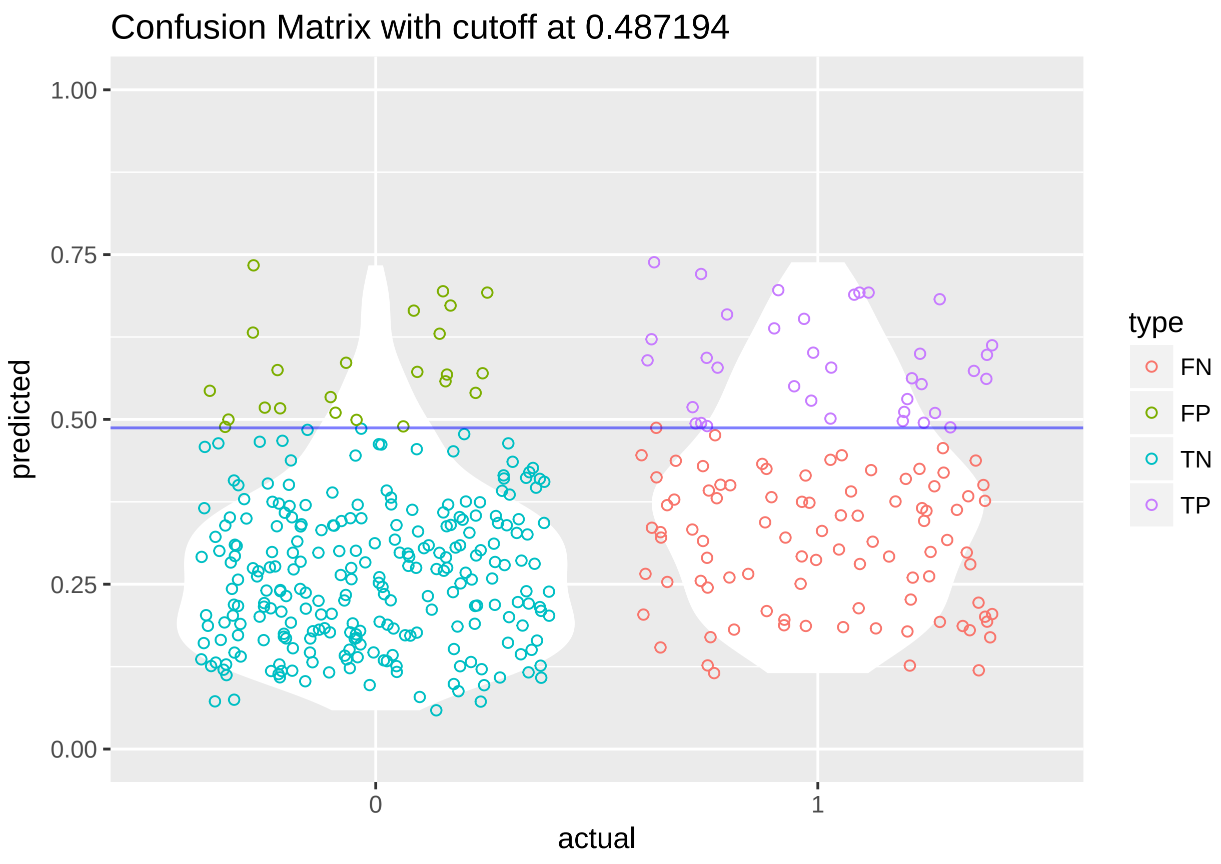
```
prediction_lgr_df2 <- predict(model_lgr_df2, data = df2, type = "response")
prediction_lgr_df2 <- if_else(prediction_lgr_df2 > cutoff , 1, 0)
confusionMatrix(data = prediction_lgr_df2,
                reference = df2$admit,
                positive = "1")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 250  91
##          1  23  36
##
##                Accuracy : 0.715
##                  95% CI : (0.668, 0.7588)
##      No Information Rate : 0.6825
##      P-Value [Acc > NIR] : 0.08878
##
##                    Kappa : 0.2325
##   Mcnemar's Test P-Value : 3.494e-10
##
##              Sensitivity : 0.2835
##              Specificity : 0.9158
##           Pos Pred Value : 0.6102
##           Neg Pred Value : 0.7331
##               Prevalence : 0.3175
##           Detection Rate : 0.0900
##     Detection Prevalence : 0.1475
##        Balanced Accuracy : 0.5996
##
##         'Positive' Class : 1
##
```

And bonus, we even gained some accuracy!

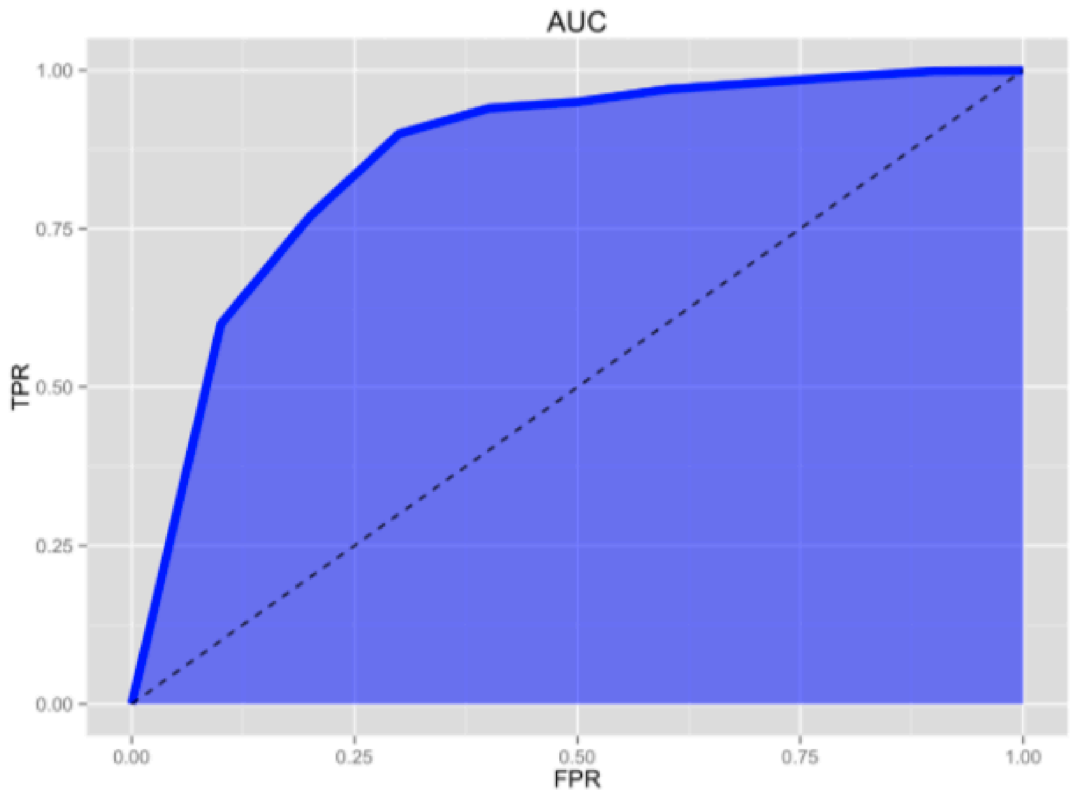
I have seen a very cool graph on this website that plots this tradeoff between specificity and sensitivity and shows how this cutoff point can enhance the understanding of the predictive power of our model.

```
# Create tibble with both prediction and actual value
cutoff = 0.487194
cutoff_plot <- tibble(predicted = predict(model_lgr_df2, data = df2, type = "response")
                      actual = as.factor(df2$admit)) %>%
                 mutate(type = if_else(predicted >= cutoff & actual == 1, "TP",
                        if_else(predicted >= cutoff & actual == 0, "FP",
                                if_else(predicted < cutoff & actual == 0, "TN", "FN")))

cutoff_plot$type <- as.factor(cutoff_plot$type)

ggplot(cutoff_plot, aes(x = actual, y = predicted, color = type)) +
  geom_violin(fill = "white", color = NA) +
  geom_jitter(shape = 1) +
  geom_hline(yintercept = cutoff, color = "blue", alpha = 0.5) +
  scale_y_continuous(limits = c(0, 1)) +
  ggtitle(paste0("Confusion Matrix with cutoff at ", cutoff))
```



Last thing ... the AUC, aka *Area Under the Curve*.
The AUC is basically the area under the ROC curve.
You can think of the AUC as sort of a holistic number that represents how well your TP and FP is looking in aggregate.

AUC=0.5 -> BAD



AUC=1 -> GOOD

So in the context of an ROC curve, the more "up and left" it looks, the larger the AUC will be and thus, the better your classifier is. Comparing AUC values is also really useful when comparing different models, as we can select the model with the high AUC value, rather than just look at the curves.

In our situation with our model `model_admission_lr`, we can find our AUC with the `ROCR` package.

```
prediction_lgr_df2 <- predict(model_lgr_df2, data = df2, type="response")
pr_admission <- prediction(prediction_lgr_df2, df2$admit)
auc_admission <- performance(pr_admission, measure = "auc")

# and to get the exact value
auc_admission@y.values[[1]]
```

```
## [1] 0.6928413
```

4.6 Example 2 - Diabetes

In our second example we will use the *Pima Indians Diabetes Data Set* that can be downloaded on the UCI Machine learning website.

We are also dropping a clean version of the file as .csv on our github dataset folder.

The data set records females patients of at least 21 years old of Pima Indian heritage.

```
df <- read_csv("dataset/diabetes.csv")
```

The dataset has 768 observations and 9 variables.

Let's rename our variables with the proper names.

```
colnames(df) <- c("pregnant", "glucose", "diastolic",
                  "triceps", "insulin", "bmi", "diabetes", "age",
                  "test")
glimpse(df)
```

```
## Observations: 768
## Variables: 9
## $ pregnant  <int> 6, 1, 8, 1, 0, 5, 3, 10, 2, 8, 4, 10, 10, 1, 5, 7, 0...
## $ glucose   <int> 148, 85, 183, 89, 137, 116, 78, 115, 197, 125, 110, ...
## $ diastolic <int> 72, 66, 64, 66, 40, 74, 50, 0, 70, 96, 92, 74, 80, 6...
## $ triceps   <int> 35, 29, 0, 23, 35, 0, 32, 0, 45, 0, 0, 0, 0, 23, 19,...
## $ insulin   <int> 0, 0, 0, 94, 168, 0, 88, 0, 543, 0, 0, 0, 0, 846, 17...
## $ bmi       <dbl> 33.6, 26.6, 23.3, 28.1, 43.1, 25.6, 31.0, 35.3, 30.5...
## $ diabetes  <dbl> 0.627, 0.351, 0.672, 0.167, 2.288, 0.201, 0.248, 0.1...
## $ age       <int> 50, 31, 32, 21, 33, 30, 26, 29, 53, 54, 30, 34, 57, ...
## $ test      <int> 1, 0, 1, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 1, 1, 1, 1, 1...
```

All variables seems to have been recorded with the appropriate type in the data frame. Let's just change the type of the response variable to factor with *positive* and *negative* levels.

```
df$test <- factor(df$test)
#levels(df$output) <- c("negative", "positive")
```

Let's do our regression on the whole dataset.

```
df2 <- df
model_lgr_df2 <- glm(test ~., data = df2, family = "binomial")
summary(model_lgr_df2)
```

```
##
## Call:
## glm(formula = test ~ ., family = "binomial", data = df2)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.5566  -0.7274  -0.4159   0.7267   2.9297
##
## Coefficients:
##               Estimate Std. Error z value Pr(>|z|)
## (Intercept) -8.4046964  0.7166359 -11.728  < 2e-16 ***
## pregnant     0.1231823  0.0320776   3.840 0.000123 ***
## glucose      0.0351637  0.0037087   9.481  < 2e-16 ***
## diastolic   -0.0132955  0.0052336  -2.540 0.011072 *
## triceps      0.0006190  0.0068994   0.090 0.928515
## insulin     -0.0011917  0.0009012  -1.322 0.186065
## bmi          0.0897010  0.0150876   5.945 2.76e-09 ***
## diabetes     0.9451797  0.2991475   3.160 0.001580 **
## age          0.0148690  0.0093348   1.593 0.111192
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 993.48  on 767  degrees of freedom
## Residual deviance: 723.45  on 759  degrees of freedom
## AIC: 741.45
##
## Number of Fisher Scoring iterations: 5
```

If we look at the z-statistic and the associated p-values, we can see that the variables `triceps`, `insulin` and `age` are not significant variables.

The logistic regression coefficients give the change in the log odds of the outcome for a one unit increase in the predictor variable. Hence, everything else being equals, any additional pregnancy increase the log odds of having diabetes (class_variable = 1) by another `0.1231`.

We can see the confidence interval for each variables using the `confint` function.

```
confint(model_lgr_df2)
```

```
## Waiting for profiling to be done...
```

```
##                    2.5 %        97.5 %
## (Intercept) -9.860319374 -7.0481062619
## pregnant     0.060918463  0.1868558244
## glucose      0.028092756  0.0426500736
## diastolic   -0.023682464 -0.0031039754
## triceps     -0.012849460  0.0142115759
## insulin     -0.002966884  0.0005821426
## bmi          0.060849478  0.1200608498
## diabetes     0.365370025  1.5386561742
## age         -0.003503266  0.0331865712
```

If we want to get the odds, we basically exponentiate the coefficients.

```
exp(coef(model_lgr_df2))
```

```
##  (Intercept)     pregnant      glucose    diastolic      triceps
## 0.0002238137 1.1310905981 1.0357892688 0.9867924485 1.0006191560
##      insulin          bmi     diabetes          age
## 0.9988090108 1.0938471417 2.5732758592 1.0149800983
```

In this way, for every additional year of age, the odds of getting diabetes (test = positive) is increasing by `1.015`.

Let's have a first look at how our model perform

```
prediction_lgr_df2 <- predict(model_lgr_df2, data = df2, type="response")
prediction_lgr_df2 <- if_else(prediction_lgr_df2 > 0.5, 1, 0)
#prediction_diabetes_lr <- factor(prediction_diabetes_lr)
#levels(prediction_diabetes_lr) <- c("negative", "positive")

table(df2$test)
```

```
##
##   0   1
## 500 268
```

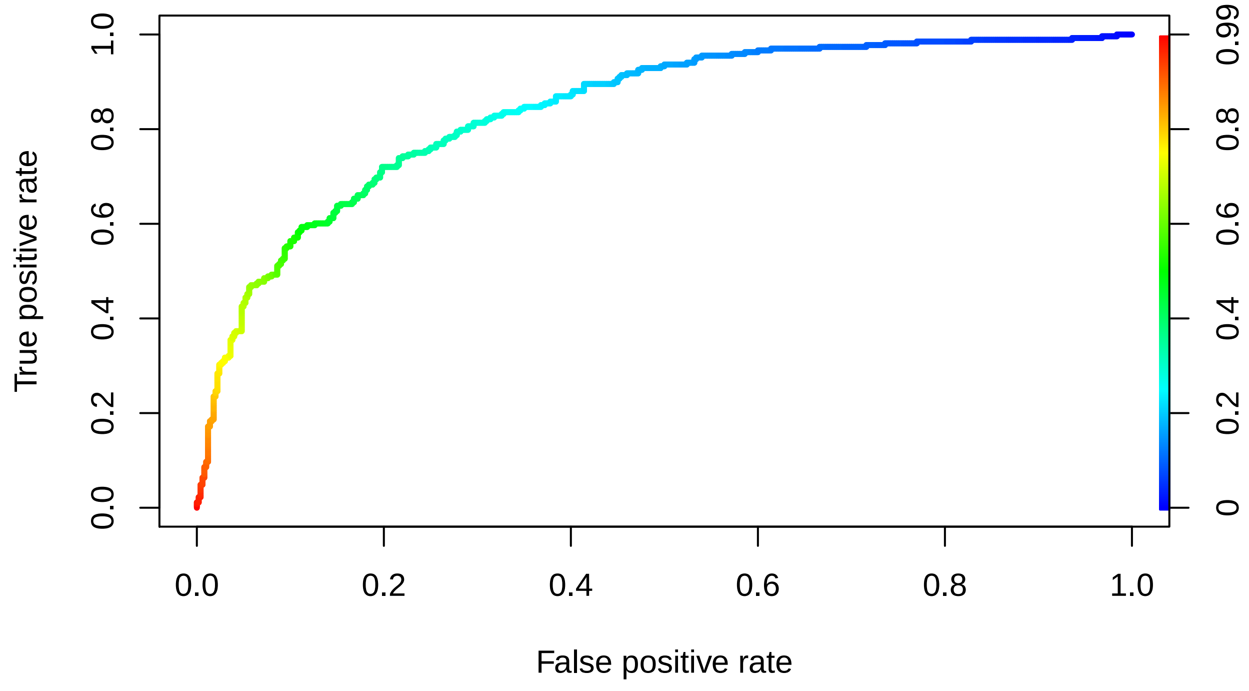
```
confusionMatrix(data = prediction_lgr_df2,
                reference = df2$test,
                positive = "1")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 445 112
##          1  55 156
##
##                Accuracy : 0.7826
##                  95% CI : (0.7517, 0.8112)
##     No Information Rate : 0.651
##     P-Value [Acc > NIR] : 1.373e-15
##
##                   Kappa : 0.4966
##  Mcnemar's Test P-Value : 1.468e-05
##
##             Sensitivity : 0.5821
##             Specificity : 0.8900
##          Pos Pred Value : 0.7393
##          Neg Pred Value : 0.7989
##              Prevalence : 0.3490
##          Detection Rate : 0.2031
##    Detection Prevalence : 0.2747
##       Balanced Accuracy : 0.7360
```

```
##
##        'Positive' Class : 1
##
```

Let's create our ROC curve

```
prediction_lgr_df2 <- predict(model_lgr_df2, data = df2, type="response")
pr_diabetes <- prediction(prediction_lgr_df2, df2$test)
prf_diabetes <- performance(pr_diabetes, measure = "tpr", x.measure = "fpr")
plot(prf_diabetes, colorize = TRUE, lwd = 3)
```

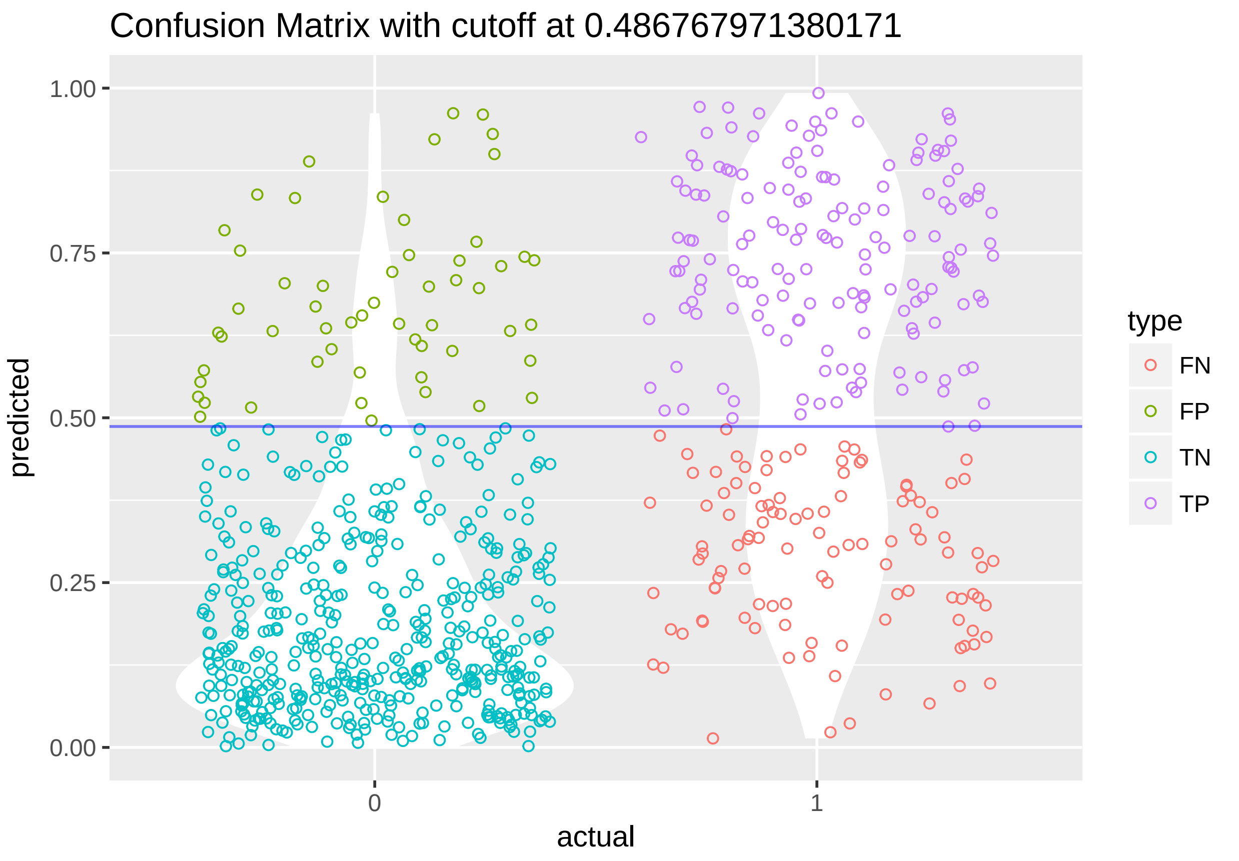


Let's find the best cutoff value for our model.

```
cost_diabetes_perf = performance(pr_diabetes, "cost")
cutoff <- pr_diabetes@cutoffs[[1]][which.min(cost_diabetes_perf@y.values[[1]])]
```

Instead of redoing the whole violin-jitter graph for our model, let's create a function so we can reuse it at a later stage.

```
violin_jitter_graph <- function(cutoff, df_predicted, df_actual){
  cutoff_tibble <- tibble(predicted = df_predicted, actual = as.factor(df_actual)) %>%
               mutate(type = if_else(predicted >= cutoff & actual == 1, "TP",
                                     if_else(predicted >= cutoff & actual == 0, "FP",
                                             if_else(predicted < cutoff & actual == 0
  cutoff_tibble$type <- as.factor(cutoff_tibble$type)

  ggplot(cutoff_tibble, aes(x = actual, y = predicted, color = type)) +
    geom_violin(fill = "white", color = NA) +
    geom_jitter(shape = 1) +
    geom_hline(yintercept = cutoff, color = "blue", alpha = 0.5) +
    scale_y_continuous(limits = c(0, 1)) +
    ggtitle(paste0("Confusion Matrix with cutoff at ", cutoff))
}


violin_jitter_graph(cutoff, predict(model_lgr_df2, data = df2, type = "response"), df2$
```



The accuracy of our model is slightly improved by using that new cutoff value.

4.6.1 Accounting for missing values

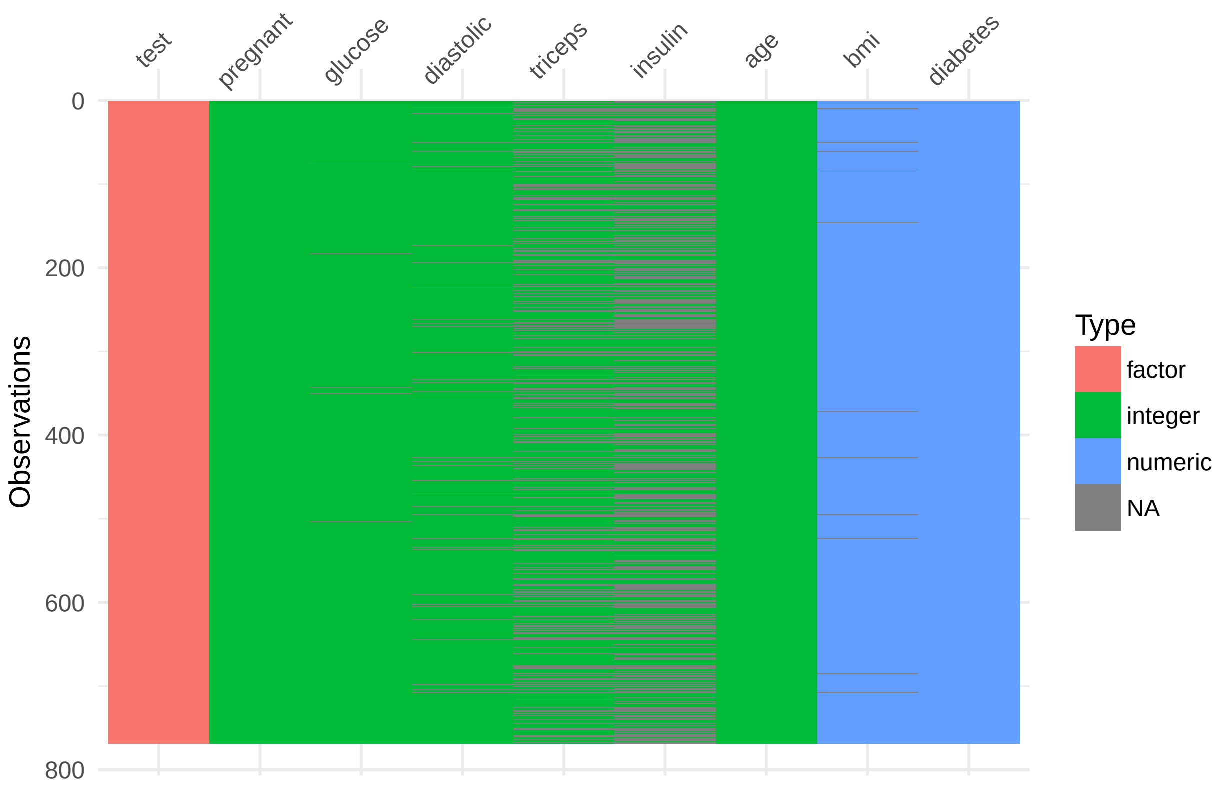
The UCI Machine Learning website note that there are no missing values on this dataset. That said, we have to be careful as there are many 0, when it is actually impossible to have such 0.
So before we keep going let's fill in these values.

The first thing to to is to change these `0` into `NA`.

```
df3 <- df2

#TODO Find a way to create a function and use map from purrr to do this
```

```
df3$glucose[df3$glucose == 0] <- NA
df3$diastolic[df3$diastolic == 0] <- NA
df3$triceps[df3$triceps == 0] <- NA
df3$insulin[df3$insulin == 0] <- NA
df3$bmi[df3$bmi == 0] <- NA
```

```
library(visdat)
vis_dat(df3)
```



There are a lot of missing values … too many of them really. If this was really life, it would be important to go back to the drawing board and redisigning the data collection phase.

```
model_lgr_df3 <- glm(test ~., data = df3, family = "binomial")
summary(model_lgr_df3)
```

```
##
## Call:
## glm(formula = test ~ ., family = "binomial", data = df3)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.7823  -0.6603  -0.3642   0.6409   2.5612
##
## Coefficients:
##                Estimate Std. Error z value Pr(>|z|)
## (Intercept) -1.004e+01  1.218e+00  -8.246  < 2e-16 ***
## pregnant     8.216e-02  5.543e-02   1.482  0.13825
## glucose      3.827e-02  5.768e-03   6.635 3.24e-11 ***
## diastolic   -1.420e-03  1.183e-02  -0.120  0.90446
## triceps      1.122e-02  1.708e-02   0.657  0.51128
## insulin     -8.253e-04  1.306e-03  -0.632  0.52757
## bmi          7.054e-02  2.734e-02   2.580  0.00989 **
## diabetes     1.141e+00  4.274e-01   2.669  0.00760 **
## age          3.395e-02  1.838e-02   1.847  0.06474 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 498.10  on 391  degrees of freedom
## Residual deviance: 344.02  on 383  degrees of freedom
##   (376 observations deleted due to missingness)
## AIC: 362.02
##
## Number of Fisher Scoring iterations: 5
```

This leads to a very different results than previously.

Let's have a look at this new model performance

```
prediction_lgr_df3 <- predict(model_lgr_df3, data = df3, type="response")
prediction_lgr_df3 <- if_else(prediction_lgr_df3 > 0.5, 1, 0)
#prediction_diabetes_lr <- factor(prediction_diabetes_lr)
#levels(prediction_diabetes_lr) <- c("negative", "positive")

table(df3$test)
```

```
##
##   0   1
## 500 268
```

```
#confusionMatrix(data = prediction_diabetes_lr2,
#                 reference = df2$test,
#                 positive = "1")
```

4.6.2 Imputting Missing Values

Now let's impute the missing values using the `simputatiion` package. A nice vignette is available here.

```
library(simputation)
df4 <- df3
df4 <- impute_lm(df3, formula = glucose ~ pregnant + diabetes + age | test)
df4 <- impute_rf(df4, formula = bmi ~ glucose + pregnant + diabetes + age | test)
df4 <- impute_rf(df4, formula = diastolic ~ bmi + glucose + pregnant + diabetes + age |
df4 <- impute_en(df4, formula = triceps ~ pregnant + bmi + diabetes + age | test)
```

```
## Warning: Package 'glmnet' is needed but not found. Returning original data
```

```
df4 <- impute_rf(df4, formula = insulin ~ . | test)

summary(df4)
```

```
##     pregnant         glucose        diastolic        triceps
##  Min.   : 0.000   Min.   : 44.00   Min.   : 24.00   Min.   : 7.00
##  1st Qu.: 1.000   1st Qu.: 99.75   1st Qu.: 64.00   1st Qu.:22.00
##  Median : 3.000   Median :117.00   Median : 72.00   Median :29.00
##  Mean   : 3.845   Mean   :121.68   Mean   : 72.36   Mean   :29.15
##  3rd Qu.: 6.000   3rd Qu.:141.00   3rd Qu.: 80.00   3rd Qu.:36.00
##  Max.   :17.000   Max.   :199.00   Max.   :122.00   Max.   :99.00
##                                                     NA's   :227
##     insulin            bmi           diabetes           age        test
##  Min.   : 14.00   Min.   :18.20   Min.   :0.0780   Min.   :21.00   0:500
##  1st Qu.: 85.05   1st Qu.:27.50   1st Qu.:0.2437   1st Qu.:24.00   1:268
##  Median :126.00   Median :32.07   Median :0.3725   Median :29.00
##  Mean   :153.59   Mean   :32.43   Mean   :0.4719   Mean   :33.24
##  3rd Qu.:188.00   3rd Qu.:36.60   3rd Qu.:0.6262   3rd Qu.:41.00
##  Max.   :846.00   Max.   :67.10   Max.   :2.4200   Max.   :81.00
##  NA's   :227
```

Ok we managed to get rid of the NAs. Let's run a last time our logistic model.

```
model_lgr_df4 <- glm(test ~ ., data = df4, family = "binomial")
summary(model_lgr_df4)
```

```
##
## Call:
## glm(formula = test ~ ., family = "binomial", data = df4)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -3.0410  -0.6503  -0.3665   0.6394   2.4823
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -9.6141841  1.0124984  -9.496  < 2e-16 ***
## pregnant     0.1279822  0.0435497   2.939 0.003295 **
## glucose      0.0348406  0.0048769   7.144 9.06e-13 ***
## diastolic   -0.0076034  0.0103592  -0.734 0.462959
## triceps      0.0070276  0.0147719   0.476 0.634257
## insulin      0.0004352  0.0012912   0.337 0.736095
## bmi          0.0836719  0.0236054   3.545 0.000393 ***
## diabetes     1.2401318  0.3562228   3.481 0.000499 ***
## age          0.0267114  0.0140647   1.899 0.057540 .
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 688.25  on 540  degrees of freedom
## Residual deviance: 470.06  on 532  degrees of freedom
##   (227 observations deleted due to missingness)
## AIC: 488.06
##
## Number of Fisher Scoring iterations: 5
```

```
prediction_lgr_df4 <- predict(model_lgr_df4, data = df4, type="response")
prediction_lgr_df4 <- if_else(prediction_lgr_df4 > 0.5, "positive", "negative")
prediction_lgr_df4 <- factor(prediction_lgr_df4)
levels(prediction_lgr_df4) <- c("negative", "positive")

#table(df4$test, prediction_lgr_df4)
#table(df4$test)
########

#confusionMatrix(data = accuracy_model_lr3,
#                 reference = df3$test,
#                 positive = "positive")
```

4.6.3 ROC and AUC

```
prediction_lgr_df4 <- predict(model_lgr_df4, data = df4, type="response")
#pr <- prediction(prediction_lgr_df4, df4$test)
#prf <- performance(pr, measure = "tpr", x.measure = "fpr")
#plot(prf)
```

Let's go back to the ideal cut off point that would balance the sensitivity and specificity.

```
#cost_diabetes_perf <- performance(pr, "cost")
#cutoff <- pr@cutoffs[[1]][which.min(cost_diabetes_perf@y.values[[1]])]
```

So for maximum accuracy, the ideal cutoff point is `0.487194`.
Let's redo our confusion matrix then and see some improvement.

```
prediction_lgr_df4 <- predict(model_lgr_df4, data = df4, type="response")
prediction_lgr_df4 <- if_else(prediction_lgr_df4 >= cutoff, "positive", "negative")

#confusionMatrix(data = accuracy_model_lr3,
#                reference = df3$test,
#                positive = "positive")
```

Another cost measure that is popular is overall accuracy. This measure optimizes the correct results, but may be skewed if there are many more negatives than positives, or vice versa. Let's get the overall accuracy for the simple predictions and plot it.

Actually the `ROCR` package can also give us a plot of accuracy for various cutoff points

```
#prediction_lgr_df4 <- performance(pr, measure = "acc")
#plot(prediction_lgr_df4)
```

Often in medical research for instance, there is a cost in having false negative is quite higher than a false positve.
Let's say the cost of missing someone having diabetes is 3 times the cost of telling someone that he has diabetes when in reality he/she doesn't.

```
#cost_diabetes_perf <- performance(pr, "cost", cost.fp = 1, cost.fn = 3)
#cutoff <- pr@cutoffs[[1]][which.min(cost_diabetes_perf@y.values[[1]])]
```

Lastly, in regards to AUC

```
#auc <- performance(pr, measure = "auc")
#auc <- auc@y.values[[1]]
#auc
```

4.7 References

- The Introduction is from the AV website
- Confusion plot. The webpage and the code
- The UCLA Institute for Digital Research and Education site where we got the dataset for our first example
- The UCI Machine learning site where we got the dataset for our second example
- Function to use ROC with ggplot2 - The Joy of Data and here as well

Chapter 5

Softmax and multinomial regressions

5.1 Multinomial Logistic Regression

5.2 References

If

Chapter 6

KNN - K Nearest Neighbour

The KNN algorithm is a robust and versatile classifier that is often used as a benchmark for more complex classifiers such as Artificial Neural Networks (ANN) and Support Vector Machines (SVM). Despite its simplicity, KNN can outperform more powerful classifiers and is used in a variety of applications.

The KNN classifier is also a non parametric and instance-based learning algorithm.

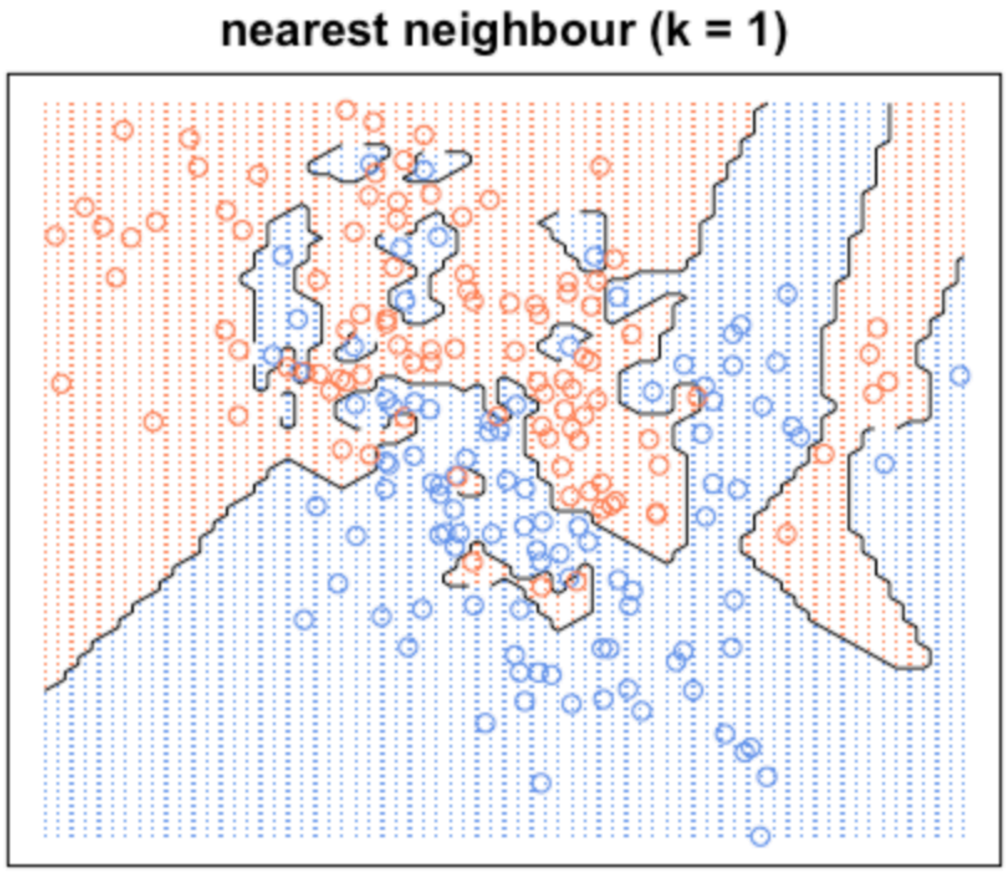
Non-parametric means it makes no explicit assumptions about the functional form of h, avoiding the dangers of mismodeling the underlying distribution of the data. For example, suppose our data is highly non-Gaussian but the learning model we choose assumes a Gaussian form. In that case, our algorithm would make extremely poor predictions.

Instance-based learning means that our algorithm doesn't explicitly learn a model (lazy learner). Instead, it chooses to memorize the training instances which are subsequently used as "knowledge" for the prediction phase. Concretely, this means that only when a query to our database is made (i.e. when we ask it to predict a label given an input), will the algorithm use the training instances to spit out an answer.

It is worth noting that the minimal training phase of KNN comes both at a memory cost, since we must store a potentially huge data set, as well as a computational cost during test time since classifying a given observation requires a run down of the whole data set. Practically speaking, this is undesirable since we usually want fast responses.

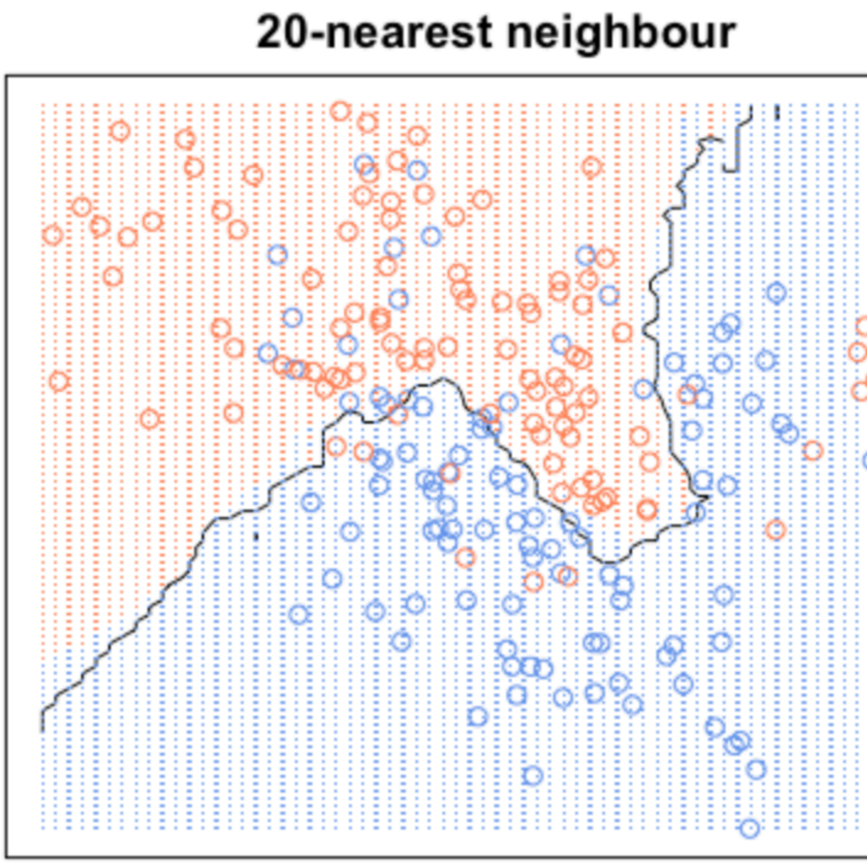
The principle behind KNN classifier (K-Nearest Neighbor) algorithm is to find K predefined number of training samples that are closest in the distance to a new point & predict a label for our new point using these samples.

When K is small, we are restraining the region of a given prediction and forcing our classifier to be "more blind" to the overall distribution. A small value for K provides the most flexible fit, which will have low bias but high variance. Graphically, our decision boundary will be



more jagged.

On the other hand, a higher K averages more voters in each prediction and hence is more resilient to outliers. Larger values of K will have smoother decision boundaries which means



lower variance but increased bias.

6.1 Example 1. Prostate Cancer dataset

```
library(tidyverse)
df <- read_csv("dataset/prostate_cancer.csv")
glimpse(df)
```

```
## Observations: 100
## Variables: 10
## $ id                <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 1...
## $ diagnosis_result  <chr> "M", "B", "M", "M", "M", "B", "M", "M", "M",...
## $ radius            <int> 23, 9, 21, 14, 9, 25, 16, 15, 19, 25, 24, 17...
## $ texture           <int> 12, 13, 27, 16, 19, 25, 26, 18, 24, 11, 21, ...
## $ perimeter         <int> 151, 133, 130, 78, 135, 83, 120, 90, 88, 84,...
## $ area              <int> 954, 1326, 1203, 386, 1297, 477, 1040, 578, ...
## $ smoothness        <dbl> 0.143, 0.143, 0.125, 0.070, 0.141, 0.128, 0....
## $ compactness       <dbl> 0.278, 0.079, 0.160, 0.284, 0.133, 0.170, 0....
## $ symmetry          <dbl> 0.242, 0.181, 0.207, 0.260, 0.181, 0.209, 0....
## $ fractal_dimension <dbl> 0.079, 0.057, 0.060, 0.097, 0.059, 0.076, 0....
```

Change the diagnosis result into a factor, then remove the `ID` variable as it does not bring anything.

```r
df$diagnosis_result <- factor(df$diagnosis_result, levels = c("B", "M"),
                              labels = c("Benign", "Malignant"))
df2 <- df %>% select(-id)

# Checking how balance is the dependend variable
prop.table(table(df2$diagnosis_result))
```

```
##
##    Benign Malignant
##      0.38      0.62
```

It is quite typical of such medical dataset to be unbalanced. We'll have to deal with it.

Like with PCA, KNN is quite sensitve to the scale of the variable. So it is important to first standardize the variables. This time we'll do this using the `preProcess` funnction of the `caret` package.

```r
library(caret)
param_preproc_df2 <- preProcess(df2[,2:9], method = c("scale", "center"))
df3_stdize <- predict(param_preproc_df2, df2[, 2:9])

summary(df3_stdize)
```

```
##      radius             texture           perimeter              area
##  Min.   :-1.60891   Min.   :-1.3923   Min.   :-1.8914   Min.   :-1.5667
##  1st Qu.:-0.99404   1st Qu.:-0.8146   1st Qu.:-0.6031   1st Qu.:-0.7073
##  Median : 0.03074   Median :-0.1406   Median :-0.1174   Median :-0.1842
##  Mean   : 0.00000   Mean   : 0.0000   Mean   : 0.0000   Mean   : 0.0000
##  3rd Qu.: 0.85057   3rd Qu.: 0.7741   3rd Qu.: 0.7379   3rd Qu.: 0.6697
##  Max.   : 1.67039   Max.   : 1.6888   Max.   : 3.1770   Max.   : 3.6756
##    smoothness         compactness         symmetry        fractal_dimension
##  Min.   :-2.23539   Min.   :-1.4507   Min.   :-1.8896   Min.   :-1.4342
##  1st Qu.:-0.63039   1st Qu.:-0.7556   1st Qu.:-0.6877   1st Qu.:-0.6981
##  Median :-0.04986   Median :-0.1341   Median :-0.1030   Median :-0.2073
##  Mean   : 0.00000   Mean   : 0.0000   Mean   : 0.0000   Mean   : 0.0000
##  3rd Qu.: 0.63312   3rd Qu.: 0.4956   3rd Qu.: 0.5142   3rd Qu.: 0.5288
##  Max.   : 2.75035   Max.   : 3.5703   Max.   : 3.6001   Max.   : 3.9639
```

We can now see that all means are centered around 0. Now we reconstruct our df with the response variable and we split the df into a training and testing set.

```r
df3_stdize <- bind_cols(diagnosis = df2$diagnosis_result, df3_stdize)

param_split<- createDataPartition(df3_stdize$diagnosis, times = 1, p = 0.8,
                                  list = FALSE)
train_df3 <- df3_stdize[param_split, ]
test_df3 <- df3_stdize[-param_split, ]

#We can check that we still have the same kind of split
prop.table(table(train_df3$diagnosis))
```

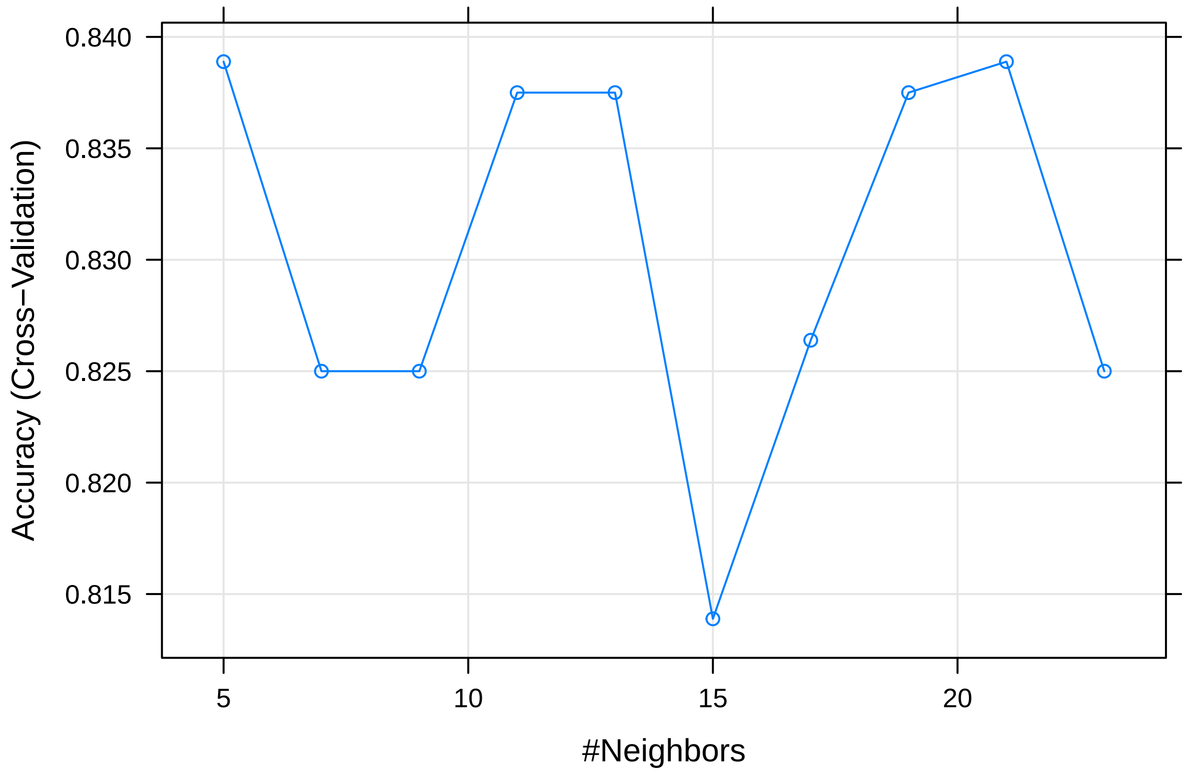
```
##
##    Benign Malignant
##  0.382716  0.617284
```

Nice to see that the proportion of *Malign* vs *Benin* has been conserved.
We use KNN with cross-validation (discussed in more details in this section 9.3 to train our model.

```r
trnctrl_df3 <- trainControl(method = "cv", number = 10)
model_knn_df3 <- train(diagnosis ~., data = train_df3, method = "knn",
                       trControl = trnctrl_df3,
                       tuneLength = 10)

model_knn_df3
```

```
## k-Nearest Neighbors
##
## 81 samples
##  8 predictors
##  2 classes: 'Benign', 'Malignant'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 73, 73, 73, 73, 73, 72, ...
## Resampling results across tuning parameters:
##
##   k   Accuracy   Kappa
##    5  0.8388889  0.6299719
##    7  0.8250000  0.6021612
##    9  0.8250000  0.5933700
##   11  0.8375000  0.6181319
##   13  0.8375000  0.6181319
##   15  0.8138889  0.5620879
##   17  0.8263889  0.5950549
##   19  0.8375000  0.6183700
##   21  0.8388889  0.6362271
##   23  0.8250000  0.6015751
##
## Accuracy was used to select the optimal model using  the largest value.
## The final value used for the model was k = 21.
```

```r
plot(model_knn_df3)
```



```r
predict_knn_df3 <- predict(model_knn_df3, test_df3)
confusionMatrix(predict_knn_df3, test_df3$diagnosis, positive = "Malignant")
```

```
## Confusion Matrix and Statistics
##
##            Reference
## Prediction  Benign Malignant
##   Benign         5         0
##   Malignant      2        12
##
##                Accuracy : 0.8947
##                  95% CI : (0.6686, 0.987)
##     No Information Rate : 0.6316
##     P-Value [Acc > NIR] : 0.01135
##
##                   Kappa : 0.7595
##  Mcnemar's Test P-Value : 0.47950
##
```

```
##             Sensitivity : 1.0000
##             Specificity : 0.7143
##          Pos Pred Value : 0.8571
##          Neg Pred Value : 1.0000
##              Prevalence : 0.6316
##          Detection Rate : 0.6316
##    Detection Prevalence : 0.7368
##       Balanced Accuracy : 0.8571
##
##        'Positive' Class : Malignant
##
```

6.2 Example 2. Wine dataset

We load the dataset and do some quick cleaning

```
df <- read_csv("dataset/Wine_UCI.csv", col_names = FALSE)
colnames(df) <- c("Origin", "Alcohol", "Malic_acid", "Ash", "Alkalinity_of_ash",
                  "Magnesium", "Total_phenols", "Flavanoids", "Nonflavonoids_phenols",
                  "Proanthocyanins", "Color_intensity", "Hue", "OD280_OD315_diluted_wine
                  "Proline")

glimpse(df)
```

```
## Observations: 178
## Variables: 14
## $ Origin                   <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
## $ Alcohol                  <dbl> 14.23, 13.20, 13.16, 14.37, 13.24, 1...
## $ Malic_acid               <dbl> 1.71, 1.78, 2.36, 1.95, 2.59, 1.76, ...
## $ Ash                      <dbl> 2.43, 2.14, 2.67, 2.50, 2.87, 2.45, ...
## $ Alkalinity_of_ash        <dbl> 15.6, 11.2, 18.6, 16.8, 21.0, 15.2, ...
## $ Magnesium                <int> 127, 100, 101, 113, 118, 112, 96, 12...
## $ Total_phenols            <dbl> 2.80, 2.65, 2.80, 3.85, 2.80, 3.27, ...
## $ Flavanoids               <dbl> 3.06, 2.76, 3.24, 3.49, 2.69, 3.39, ...
## $ Nonflavonoids_phenols    <dbl> 0.28, 0.26, 0.30, 0.24, 0.39, 0.34, ...
## $ Proanthocyanins          <dbl> 2.29, 1.28, 2.81, 2.18, 1.82, 1.97, ...
## $ Color_intensity          <dbl> 5.64, 4.38, 5.68, 7.80, 4.32, 6.75, ...
## $ Hue                      <dbl> 1.04, 1.05, 1.03, 0.86, 1.04, 1.05, ...
## $ OD280_OD315_diluted_wines <dbl> 3.92, 3.40, 3.17, 3.45, 2.93, 2.85, ...
## $ Proline                  <int> 1065, 1050, 1185, 1480, 735, 1450, 1...
```

The origin is our dependent variable. Let's make it a factor.

```
df$Origin <- as.factor(df$Origin)
```

```r
#Let's check our explained variable distribution of origin
round(prop.table(table(df$Origin)), 2)
```

```
##
##    1    2    3
## 0.33 0.40 0.27
```

That's nice, our explained variable is almost equally distributed with the 3 set of origin.

```r
# Let's also check if we have any NA values
summary(df)
```

```
##  Origin    Alcohol        Malic_acid         Ash         Alkalinity_of_ash
##  1:59   Min.   :11.03   Min.   :0.740   Min.   :1.360   Min.   :10.60
##  2:71   1st Qu.:12.36   1st Qu.:1.603   1st Qu.:2.210   1st Qu.:17.20
##  3:48   Median :13.05   Median :1.865   Median :2.360   Median :19.50
##         Mean   :13.00   Mean   :2.336   Mean   :2.367   Mean   :19.49
##         3rd Qu.:13.68   3rd Qu.:3.083   3rd Qu.:2.558   3rd Qu.:21.50
##         Max.   :14.83   Max.   :5.800   Max.   :3.230   Max.   :30.00
##    Magnesium      Total_phenols     Flavanoids    Nonflavonoids_phenols
##  Min.   : 70.00   Min.   :0.980   Min.   :0.340   Min.   :0.1300
##  1st Qu.: 88.00   1st Qu.:1.742   1st Qu.:1.205   1st Qu.:0.2700
##  Median : 98.00   Median :2.355   Median :2.135   Median :0.3400
##  Mean   : 99.74   Mean   :2.295   Mean   :2.029   Mean   :0.3619
##  3rd Qu.:107.00   3rd Qu.:2.800   3rd Qu.:2.875   3rd Qu.:0.4375
##  Max.   :162.00   Max.   :3.880   Max.   :5.080   Max.   :0.6600
##  Proanthocyanins Color_intensity       Hue
##  Min.   :0.410   Min.   : 1.280   Min.   :0.4800
##  1st Qu.:1.250   1st Qu.: 3.220   1st Qu.:0.7825
##  Median :1.555   Median : 4.690   Median :0.9650
##  Mean   :1.591   Mean   : 5.058   Mean   :0.9574
##  3rd Qu.:1.950   3rd Qu.: 6.200   3rd Qu.:1.1200
##  Max.   :3.580   Max.   :13.000   Max.   :1.7100
##  OD280_OD315_diluted_wines    Proline
##  Min.   :1.270             Min.   : 278.0
##  1st Qu.:1.938             1st Qu.: 500.5
##  Median :2.780             Median : 673.5
##  Mean   :2.612             Mean   : 746.9
##  3rd Qu.:3.170             3rd Qu.: 985.0
##  Max.   :4.000             Max.   :1680.0
```

Here we noticed that the range of values in our variable is quite wide. It means our data will need to be standardize. We also note that we no "NA" values. That's quite a nice surprise!

6.2.1 Understand the data

We first slide our data in a training and testing set.

```
df2 <- df
param_split_df2 <- createDataPartition(df2$Origin, p = 0.75, list = FALSE)

train_df2 <- df2[param_split_df2, ]
test_df2 <- df2[-param_split_df2, ]
```

The great with caret is we can standardize our data in the the training phase.

6.2.1.1 Model the data

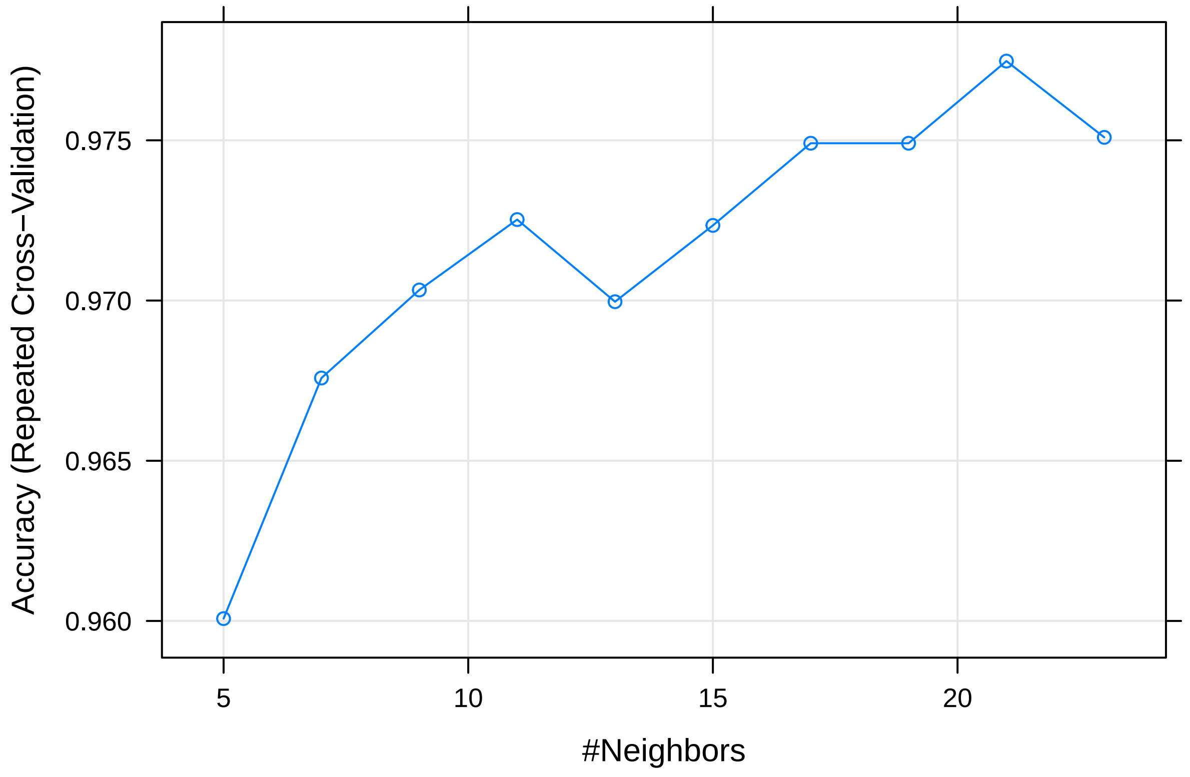
Let's keep using `caret` for our training.

```
trnctrl_df2 <- trainControl(method = "repeatedcv", number = 10, repeats = 3)
model_knn_df2 <- train(Origin ~., data = train_df2, method = "knn",
                       trControl = trnctrl_df2,
                       preProcess = c("center", "scale"),
                       tuneLength = 10)
```

```
model_knn_df2
```

```
## k-Nearest Neighbors
##
## 135 samples
##  13 predictors
##   3 classes: '1', '2', '3'
##
## Pre-processing: centered (13), scaled (13)
## Resampling: Cross-Validated (10 fold, repeated 3 times)
## Summary of sample sizes: 122, 121, 121, 121, 122, 121, ...
## Resampling results across tuning parameters:
##
##   k   Accuracy   Kappa
##    5  0.9600733  0.9392079
##    7  0.9675824  0.9508517
##    9  0.9703297  0.9549689
##   11  0.9725275  0.9584661
##   13  0.9699634  0.9546670
##   15  0.9723443  0.9583141
##   17  0.9749084  0.9620538
##   19  0.9749084  0.9620538
##   21  0.9774725  0.9659932
##   23  0.9750916  0.9624313
##
## Accuracy was used to select the optimal model using  the largest value.
## The final value used for the model was k = 21.
```

```
plot(model_knn_df2)
```



Let's use our model to make our prediction

```
prediction_knn_df2 <- predict(model_knn_df2, newdata = test_df2)

confusionMatrix(prediction_knn_df2, reference = test_df2$Origin)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  1  2  3
##          1 14  2  0
##          2  0 13  0
##          3  0  2 12
##
## Overall Statistics
##
##                Accuracy : 0.907
##                  95% CI : (0.7786, 0.9741)
##     No Information Rate : 0.3953
##     P-Value [Acc > NIR] : 3.376e-12
##
##                   Kappa : 0.8608
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 1 Class: 2 Class: 3
## Sensitivity            1.0000   0.7647   1.0000
## Specificity            0.9310   1.0000   0.9355
## Pos Pred Value         0.8750   1.0000   0.8571
## Neg Pred Value         1.0000   0.8667   1.0000
## Prevalence             0.3256   0.3953   0.2791
## Detection Rate         0.3256   0.3023   0.2791
## Detection Prevalence   0.3721   0.3023   0.3256
## Balanced Accuracy      0.9655   0.8824   0.9677
```

6.3 References

- KNN R, K-Nearest neighbor implementation in R using caret package. Here
- A complete guide to KNN. Here

Chapter 7

Principal Component Analysis

To create a predictive model based on regression we like to have as many relevant predictors as possible. The whole difficulty resides in finding *relevant* predictors. For predictors to be relevant, they should explain the variance of the dependent variable.
Too many predictors (high dimensionality) and we take the risk of over-fitting.

The intuition of Principal Component Analysis is to find new combination of variables which form larger variances. Why are larger variances important? This is a similar concept of entropy in information theory. Let's say you have two variables. One of them (Var 1) forms N(1, 0.01) and the other (Var 2) forms N(1, 1). Which variable do you think has more information? Var 1 is always pretty much 1 whereas Var 2 can take a wider range of values, like 0 or 2. Thus, Var 2 has more chances to have various values than Var 1, which means Var 2's entropy is larger than Var 1's. Thus, we can say Var 2 contains more information than Var 1.

PCA tries to find linear combination of the variables which contain much information by looking at the variance. This is why the standard deviation is one of the important metrics to determine the number of new variables in PCA. Another interesting aspect of the new variables derived by PCA is that all new variables are orthogonal. You can think that PCA is rotating and translating the data such that the first axis contains the most information, and the second has the second most information, and so forth.

Principal Component Analysis (PCA) is a feature extraction methods that use orthogonal linear projections to capture the underlying variance of the data. When PCR compute the principle components is not looking at the response but only at the predictors (by looking for a linear combination of the predictors that has the highest variance). It makes the assumption that the linear combination of the predictors that has the highest variance is associated with the response.

The algorithm when applied linearly transforms m-dimensional input space to n-dimensional ($n < m$) output space, with the objective to minimize the amount of information/variance lost by discarding (m-n) dimensions. PCA allows us to discard the variables/features that have less variance.

When choosing the principal component, we assume that the regression plane varies along the line and doesn't vary in the other orthogonal direction. By choosing one component and not the other, we're ignoring the second direction.

PCR looks in the direction of variation of the predictors to find the places where the responses is most likely to vary.

Some of the most notable advantages of performing PCA are the following:

- Dimensionality reduction
- Avoidance of multicollinearity between predictors. Variables are orthogonal, so including, say, PC9 in the model has no bearing on, say, PC3
- Variables are ordered in terms of standard error. Thus, they also tend to be ordered in terms of statistical significance
- Overfitting mitigation

The primary disadvantage is that this model is far more difficult to interpret than a regular logistic regression model

With principal components regression, the new transformed variables (the principal components) are calculated in a totally **unsupervised** way:

- the response Y is not used to help determine the principal component directions).
- the response does not supervise the identification of the principal components.
- PCR just looks at the x variables

The PCA method can dramatically improve estimation and insight in problems where multicollinearity is a large problem – as well as aid in detecting it.

7.1 PCA on an easy example.

Let's say we asked 16 participants four questions (on a 7 scale) about what they care about when choosing a new computer, and got the results like this.

```
Price <- c(6,7,6,5,7,6,5,6,3,1,2,5,2,3,1,2)
Software <- c(5,3,4,7,7,4,7,5,5,3,6,7,4,5,6,3)
Aesthetics <- c(3,2,4,1,5,2,2,4,6,7,6,7,5,6,5,7)
Brand <- c(4,2,5,3,5,3,1,4,7,5,7,6,6,5,5,7)
buy_computer <- tibble(Price, Software, Aesthetics, Brand)
```

Let's go on with the PCA. `princomp` is part of the *stats* package.

```
pca_buycomputer <- prcomp(buy_computer, scale = TRUE, center = TRUE)
names(pca_buycomputer)
```

```
## [1] "sdev"     "rotation" "center"   "scale"    "x"
```

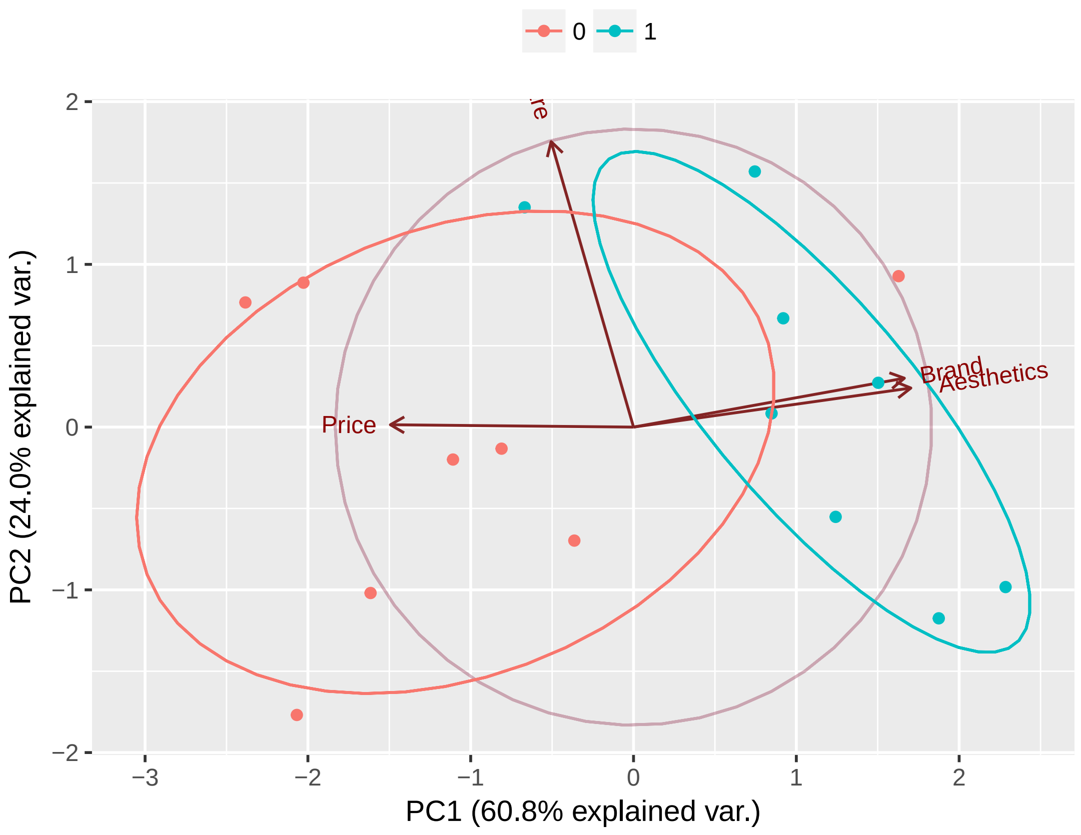
```r
print(pca_buycomputer)
```

```
## Standard deviations (1, .., p=4):
## [1] 1.5589391 0.9804092 0.6816673 0.3792578
##
## Rotation (n x k) = (4 x 4):
##                   PC1        PC2        PC3         PC4
## Price      -0.5229138 0.00807487 -0.8483525  0.08242604
## Software   -0.1771390 0.97675554  0.1198660  0.01423081
## Aesthetics  0.5965260 0.13369503 -0.2950727  0.73431229
## Brand       0.5825287 0.16735905 -0.4229212 -0.67363855
```

```r
summary(pca_buycomputer, loadings = TRUE)
```

```
## Importance of components:
##                           PC1    PC2    PC3     PC4
## Standard deviation     1.5589 0.9804 0.6817 0.37926
## Proportion of Variance 0.6076 0.2403 0.1162 0.03596
## Cumulative Proportion  0.6076 0.8479 0.9640 1.00000
```

```r
OS <- c(0,0,0,0,1,0,0,0,1,1,0,1,1,1,1,1)
library(ggbiplot)
g <- ggbiplot(pca_buycomputer, obs.scale = 1, var.scale = 1, groups = as.character(OS),
              ellipse = TRUE, circle = TRUE)
g <- g + scale_color_discrete(name = '')
g <- g + theme(legend.direction = 'horizontal',
               legend.position = 'top')
print(g)
```



Remember that one of the disadventage of PCA is how difficult it is to interpret the model (ie. what does the PC1 is representing, what does PC2 is representing, etc.). The **biplot** graph help somehow to overcome that.

In the above graph, one can see that `Brand`and `Aesthetic` explain most of the variance in the new predictor PC1 while `Software` explain most of the variance in the new predictor PC2. It is also to be noted that `Brand` and `Aesthetic` are quite highly correlated.

Once you have done the analysis with PCA, you may want to look into whether the new variables can predict some phenomena well. This is kinda like machine learning: Whether features can classify the data well. Let's say you have asked the participants one more thing, which OS they are using (Windows or Mac) in your survey, and the results are like this.

```
OS <- c(0,0,0,0,1,0,0,0,1,1,0,1,1,1,1,1)
# Let's test our model
model1 <- glm(OS ~ pca_buycomputer$x[,1] + pca_buycomputer$x[,2], family = binomial)
summary(model1)
```

```
##
## Call:
## glm(formula = OS ~ pca_buycomputer$x[, 1] + pca_buycomputer$x[,
##     2], family = binomial)
##
## Deviance Residuals:
```

```
##     Min       1Q   Median       3Q      Max
## -2.4485  -0.4003   0.1258   0.5652   1.2814
##
## Coefficients:
##                        Estimate Std. Error z value Pr(>|z|)
## (Intercept)             -0.2138     0.7993  -0.268   0.7891
## pca_buycomputer$x[, 1]   1.5227     0.6621   2.300   0.0215 *
## pca_buycomputer$x[, 2]   0.7337     0.9234   0.795   0.4269
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 22.181  on 15  degrees of freedom
## Residual deviance: 11.338  on 13  degrees of freedom
## AIC: 17.338
##
## Number of Fisher Scoring iterations: 5
```

Let's see how well this model predicts the kind of OS. You can use fitted() function to see the prediction.

```
fitted(model1)
```

```
##           1           2           3           4           5           6
## 0.114201733 0.009372181 0.217716320 0.066009817 0.440016243 0.031640529
##           7           8           9          10          11          12
## 0.036189119 0.175766013 0.906761064 0.855587371 0.950088045 0.888272270
##          13          14          15          16
## 0.781098710 0.757499202 0.842557931 0.927223453
```

These values represent the probabilities of being 1. For example, we can expect 11% chance that Participant 1 is using OS 1 based on the variable derived by PCA. Thus, in this case, Participant 1 is more likely to be using OS 0, which agrees with the survey response. In this way, PCA can be used with regression models for calculating the probability of a phenomenon or making a prediction.

I have tried to do the same with scaling the data using `scale(x)` and it changed absolutely nothing.

In general, the data will tend to follow the 80/20 rule. Most of the variance (interesting part of data) will be explained by a very small number of principal components. You might be able to explain 95% of the variance in your dataset using only 10% of the original number of attributes. However, this is entirely dependent on the dataset. Often, a good rule of thumb is to identify the principal components that explain 99% of the variance in the data.

7.2 References.

Here are the articles I have consulted for this research.

- Principal Component Analysis (PCA)
- Principal Component Analysis using R
- Computing and visualizing PCA in R This is where we learned about the ‘ggbiplot
- Practical Guide to Principal Component Analysis (PCA) in R & Python
- Performing Principal Components Regression (PCR) in R
- Data Mining - Principal Component (Analysis|Regression) (PCA)
- PRINCIPAL COMPONENT ANALYSIS IN R A really nice explanation on the difference between the main packages doing PCA such as `svd`, `princomp`and `prcomp`. In R there are two general methods to perform PCA without any missing values: The spectral decomposition method of analysis examines the covariances and correlations between variables, whereas the singular value decomposition method looks at the covariances and correlations among the samples. While both methods can easily be performed within R, the singular value decomposition method is the preferred analysis for numerical accuracy.

Although principal component analysis assumes multivariate normality, this is not a very strict assumption, especially when the procedure is used for data reduction or exploratory purposes. Undoubtedly, the correlation and covariance matrices are better measures of similarity if the data is normal, and yet, PCA is often unaffected by mild violations. However, if the new components are to be used in further analyses, such as regression analysis, normality of the data might be more important.

Chapter 8

Trees, Random forests and Classification

8.1 Introduction

Classification trees are non-parametric methods to recursively partition the data into more “pure” nodes, based on splitting rules.

Logistic regression vs Decision trees. It is dependent on the type of problem you are solving. Let’s look at some key factors which will help you to decide which algorithm to use:

- If the relationship between dependent & independent variable is well approximated by a linear model, linear regression will outperform tree based model.
- If there is a high non-linearity & complex relationship between dependent & independent variables, a tree model will outperform a classical regression method.
- If you need to build a model which is easy to explain to people, a decision tree model will always do better than a linear model. Decision tree models are even simpler to interpret than linear regression!

The 2 main disadventages of Decision trees: **Over fitting**: Over fitting is one of the most practical difficulty for decision tree models. This problem gets solved by setting constraints on model parameters and pruning (discussed in detailed below).

Not fit for continuous variables: While working with continuous numerical variables, decision tree looses information when it categorizes variables in different categories.

Decision trees use multiple algorithms to decide to split a node in two or more sub-nodes. The creation of sub-nodes increases the homogeneity of resultant sub-nodes. In other words, we can say that purity of the node increases with respect to the target variable. Decision tree splits the nodes on all available variables and then selects the split which results in most homogeneous sub-nodes.

8.2 First example.

Let's do a CART on the iris dataset. This is the `Hello World!` of CART.

```
library(rpart)
library(rpart.plot)
data("iris")
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
## $ Species    : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```
table(iris$Species)
```

```
##
##     setosa versicolor  virginica
##         50         50         50
```

```
tree <- rpart(Species ~., data = iris, method = "class")
tree
```

```
## n= 150
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
## 1) root 150 100 setosa (0.33333333 0.33333333 0.33333333)
##   2) Petal.Length< 2.45 50   0 setosa (1.00000000 0.00000000 0.00000000) *
##   3) Petal.Length>=2.45 100  50 versicolor (0.00000000 0.50000000 0.50000000)
##     6) Petal.Width< 1.75 54   5 versicolor (0.00000000 0.90740741 0.09259259) *
##     7) Petal.Width>=1.75 46   1 virginica (0.00000000 0.02173913 0.97826087) *
```

The method-argument can be switched according to the type of the response variable. It is `class` for categorial, `anova` for numerical, `poisson` for count data and 'exp for survival data.

Important Terminology related to Decision Trees

Root Node: It represents entire population or sample and this further gets divided into two or more homogeneous sets.

Splitting: It is a process of dividing a node into two or more sub-nodes.

Decision Node: When a sub-node splits into further sub-nodes, then it is called decision node.

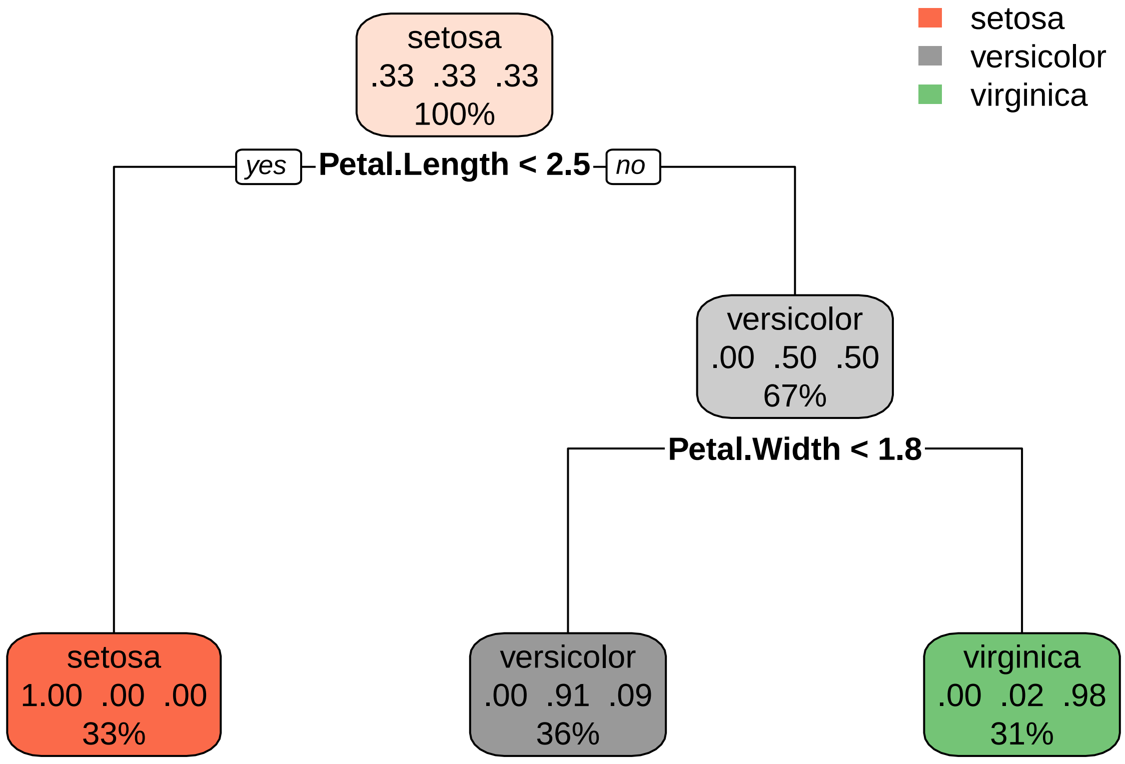
Leaf/ Terminal Node: Nodes do not split is called Leaf or Terminal node.

Pruning: When we remove sub-nodes of a decision node, this process is called pruning. You can say opposite process of splitting.

Branch / Sub-Tree: A sub section of entire tree is called branch or sub-tree.

Parent and Child Node: A node, which is divided into sub-nodes is called parent node of sub-nodes where as sub-nodes are the child of parent node.

```r
rpart.plot(tree)
```



This is a model with a **multi-class response**. Each node shows

- the predicted class (setosa, versicolor, virginica),
- the predicted probability of each class,
- the percentage of observations in the node

```r
table(iris$Species, predict(tree, type = "class"))
```

```
##
##              setosa versicolor virginica
##   setosa         50          0         0
##   versicolor      0         49         1
##   virginica       0          5        45
```

8.3 Second Example.

Data set is the titanic. This is a model with a **binary response**.

```
data("ptitanic")
str(ptitanic)
```

```
## 'data.frame':    1309 obs. of  6 variables:
##  $ pclass  : Factor w/ 3 levels "1st","2nd","3rd": 1 1 1 1 1 1 1 1 1 1 ...
##  $ survived: Factor w/ 2 levels "died","survived": 2 2 1 1 1 2 2 1 2 1 ...
##  $ sex     : Factor w/ 2 levels "female","male": 1 2 1 2 1 2 1 2 1 2 ...
##  $ age     :Class 'labelled'  atomic [1:1309] 29 0.917 2 30 25 ...
##   .. ..- attr(*, "units")= chr "Year"
##   .. ..- attr(*, "label")= chr "Age"
##  $ sibsp   :Class 'labelled'  atomic [1:1309] 0 1 1 1 1 0 1 0 2 0 ...
##   .. ..- attr(*, "label")= chr "Number of Siblings/Spouses Aboard"
##  $ parch   :Class 'labelled'  atomic [1:1309] 0 2 2 2 2 0 0 0 0 0 ...
##   .. ..- attr(*, "label")= chr "Number of Parents/Children Aboard"
```

```
ptitanic$age <- as.numeric(ptitanic$age)
ptitanic$sibsp <- as.integer(ptitanic$sibsp)
ptitanic$parch <- as.integer(ptitanic$parch)
```

Actually we can make the table more relevant.

```
round(prop.table(table(ptitanic$sex, ptitanic$survived), 1), 2)
```

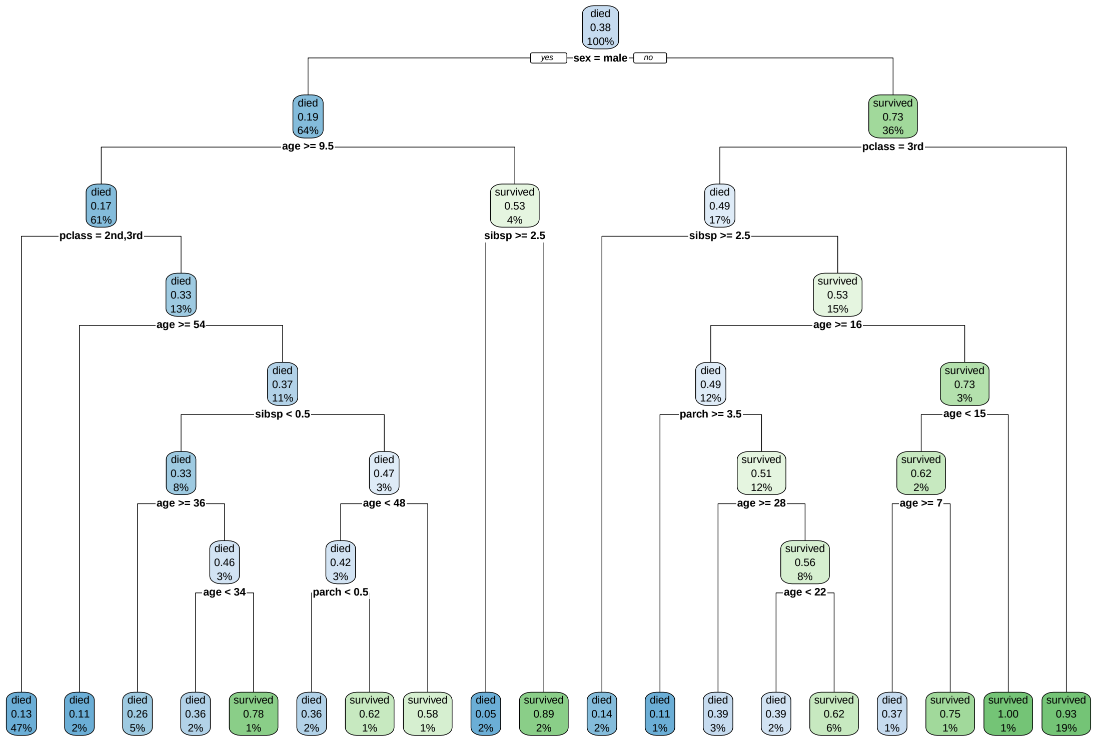
```
##
##          died survived
##   female 0.27     0.73
##   male   0.81     0.19
```

One can see here that the sum of the percentage add to 1 horizontally. If one want to make it vertically, we use *2*.

You can find the default limits by typing ?rpart.control. The first one we want to unleash is the `cp` parameter, this is the metric that stops splits that aren't deemed important enough. The other one we want to open up is `minsplit` which governs how many passengers must sit in a bucket before even looking for a split.

By putting a very low `cp` we are asking to have a very deep tree. The idea is that we prune it later. So in this first regression on `ptitanic` we'll set a very low cp.

```
library(rpart)
library(rpart.plot)
set.seed(123)
tree <- rpart(survived ~ ., data = ptitanic, cp=0.00001)
rpart.plot(tree)
```



Each node shows

- the predicted class (died or survived),
- the predicted probability of survival,
- the percentage of observations in the node.

Let's do a confusion matrix based on this tree.

```r
conf.matrix <- round(prop.table(table(ptitanic$survived, predict(tree, type="class")),
rownames(conf.matrix) <- c("Actually died", "Actually Survived")
colnames(conf.matrix) <- c("Predicted dead", "Predicted Survived")
conf.matrix
```

```
##
##                      Predicted dead Predicted Survived
##   Actually died                0.83               0.16
##   Actually Survived            0.17               0.84
```

Let's learn a bit more about trees. By using the `name` function, one can see all the object inherent to the `tree` function.
A few intersting ones. The '$where component indicates to which leaf the different observations have been assigned.

```r
names(tree)
```

```
##  [1] "frame"               "where"               "call"
##  [4] "terms"               "cptable"             "method"
##  [7] "parms"               "control"             "functions"
## [10] "numresp"             "splits"               "csplit"
## [13] "variable.importance" "y"                    "ordered"
```

How to prune a tree? We want the cp value (with a simpler tree) that minimizes the xerror. So you need to find the lowest Cross-Validation Error. 2 ways to do this. Either the `plotcp` or the `printcp` functions. The `plotcp` is a visual representation of `printcp` function.

The problem with reducing the ‘xerror is that the cross-validation error is a random quantity. There is no guarantee that if we were to fit the sequence of trees again using a different random seed that the same tree would minimize the cross-validation error.
A more robust alternative to minimum cross-validation error is to use the one standard deviation rule: choose the smallest tree whose cross-validation error is within one standard error of the minimum. Depending on how we define this there are two possible choices. The first tree whose point estimate of the cross-validation error falls within the ± 1 xstd of the minimum. On the other hand the standard error lower limit of the tree of size three is within + 1 xstd of the minimum.

Either of these is a reasonable choice, but insisting that the point estimate itself fall within the standard error limits is probably the more robust solution.

As discussed earlier, the technique of setting constraint is a greedy-approach. In other words, it will check for the best split instantaneously and move forward until one of the specified stopping condition is reached. Let’s consider the following case when you’re driving: There are 2 lanes: A lane with cars moving at 80km/h A lane with trucks moving at 30km/h At this instant, you are a car in the fast lane and you have 2 choices: Take a left and overtake the other 2 cars quickly Keep moving in the present lane Lets analyze these choice. In the former choice, you’ll immediately overtake the car ahead and reach behind the truck and start moving at 30 km/h, looking for an opportunity to move back right. All cars originally behind you move ahead in the meanwhile. This would be the optimum choice if your objective is to maximize the distance covered in next say 10 seconds. In the later choice, you sale through at same speed, cross trucks and then overtake maybe depending on situation ahead. Greedy you!

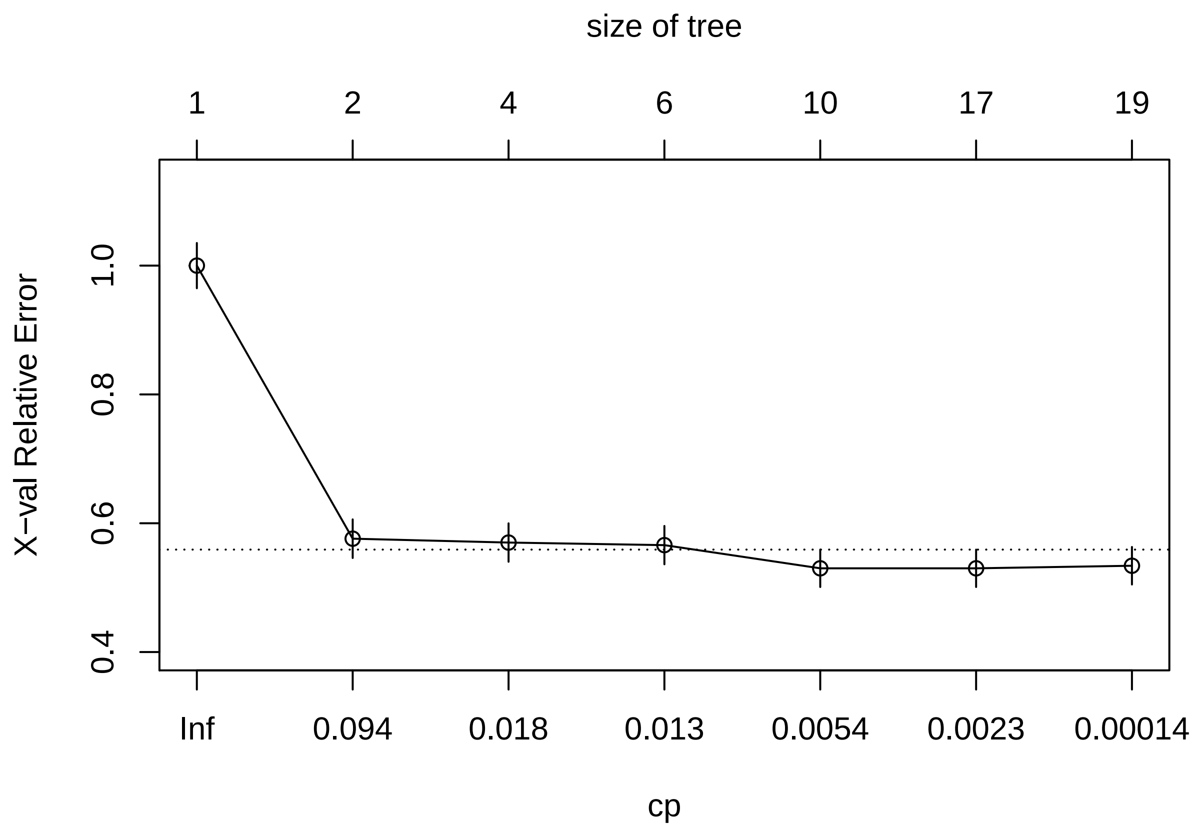
This is exactly the difference between normal decision tree & pruning. A decision tree with constraints won’t see the truck ahead and adopt a greedy approach by taking a left. On the other hand if we use pruning, we in effect look at a few steps ahead and make a choice. So we know pruning is better.

```r
printcp(tree)
```

```
##
## Classification tree:
## rpart(formula = survived ~ ., data = ptitanic, cp = 1e-05)
##
## Variables actually used in tree construction:
## [1] age    parch  pclass sex    sibsp
##
## Root node error: 500/1309 = 0.38197
```

```
##
## n= 1309
##
##          CP nsplit rel error xerror     xstd
## 1 0.4240000      0     1.000  1.000 0.035158
## 2 0.0210000      1     0.576  0.576 0.029976
## 3 0.0150000      3     0.534  0.570 0.029863
## 4 0.0113333      5     0.504  0.566 0.029787
## 5 0.0025714      9     0.458  0.530 0.029076
## 6 0.0020000     16     0.440  0.530 0.029076
## 7 0.0000100     18     0.436  0.534 0.029157
```

```
plotcp(tree)
```

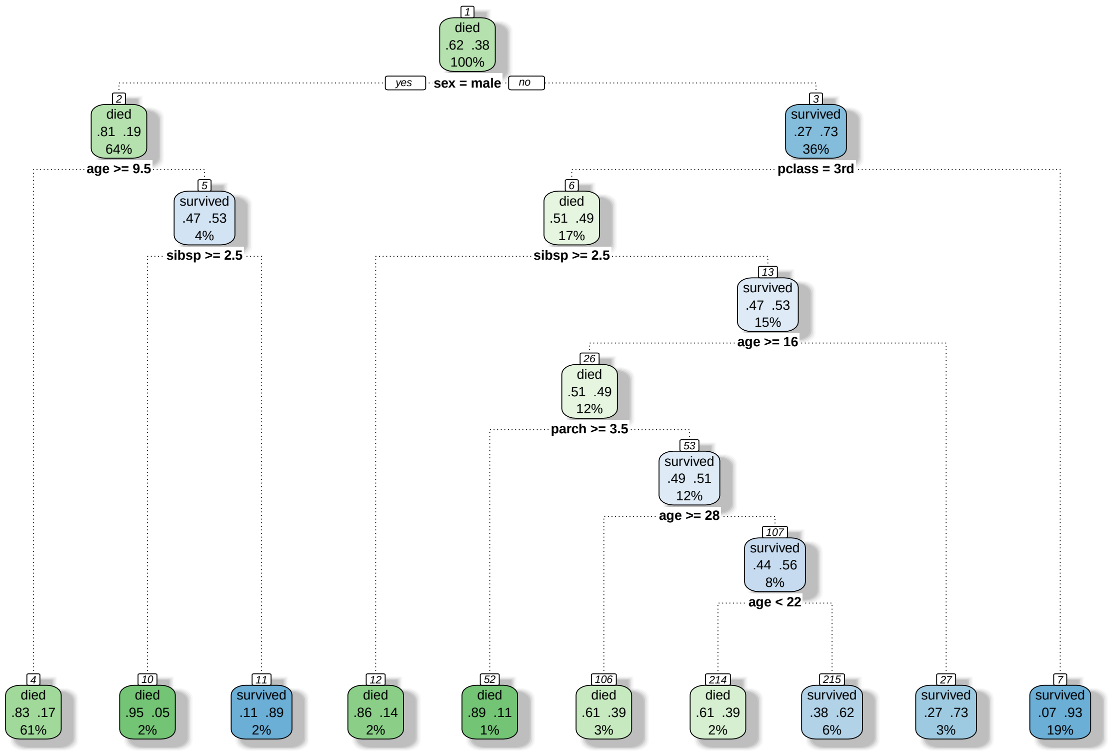


```
tree$cptable[which.min(tree$cptable[,"xerror"]),"CP"]
```

```
## [1] 0.002571429
```

See if we can prune slightly the tree

```
bestcp <- tree$cptable[which.min(tree$cptable[,"xerror"]),"CP"]
tree.pruned <- prune(tree, cp = bestcp)

#this time we add a few arguments to add some mojo to our graphed tree.
#Actually this will give us a very similar graphed tree as rattle (and we like that gr
rpart.plot(tree.pruned, extra=104, box.palette="GnBu",
                branch.lty=3, shadow.col="gray", nn=TRUE)
```

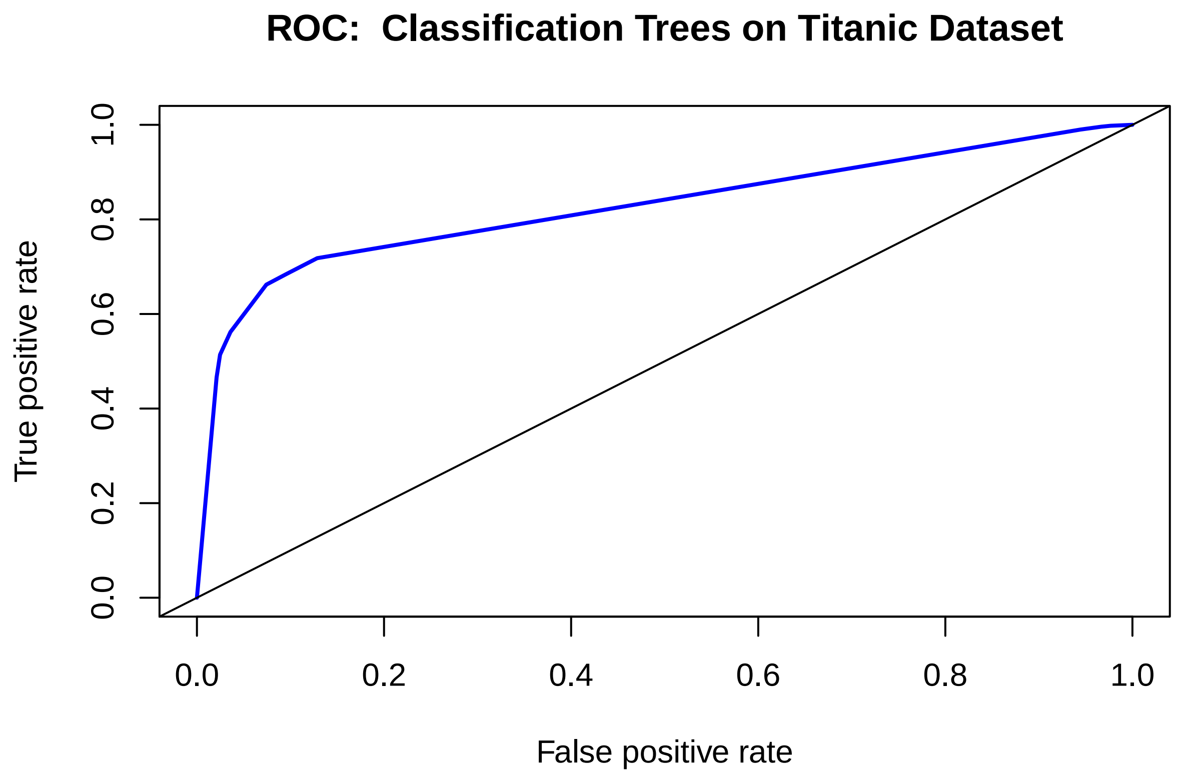


```
conf.matrix <- round(prop.table(table(ptitanic$survived, predict(tree.pruned, type="cla
rownames(conf.matrix) <- c("Actually died", "Actually Survived")
colnames(conf.matrix) <- c("Predicted dead", "Predicted Survived")
conf.matrix
```

```
##
##                        Predicted dead Predicted Survived
##   Actually died                  0.57               0.05
##   Actually Survived              0.13               0.25
```

Another way to check the output of the classifier is with a ROC (Receiver Operating Characteristics) Curve. This plots the true positive rate against the false positive rate, and gives us a visual feedback as to how well our model is performing. The package we will use for this is ROCR.

```
library(ROCR)
fit.pr = predict(tree.pruned, type="prob")[,2]
fit.pred = prediction(fit.pr, ptitanic$survived)
fit.perf = performance(fit.pred,"tpr","fpr")
plot(fit.perf,lwd=2,col="blue",
     main="ROC:  Classification Trees on Titanic Dataset")
abline(a=0,b=1)
```



Ordinarily, using the confusion matrix for creating the ROC curve would give us a single point (as it is based off True positive rate vs false positive rate). What we do here is ask the prediction algorithm to give class probabilities to each observation, and then we plot the performance of the prediction using class probability as a cutoff. This gives us the "smooth" ROC curve.

8.4 How does a tree decide where to split?

A bit more theory, before we go further. This part has been taken from this great tutorial.

8.5 Third example.

The dataset I will be using for this third example is the "Adult" dataset hosted on UCI's Machine Learning Repository. It contains approximately 32000 observations, with 15 variables. The dependent variable that in all cases we will be trying to predict is whether or not an "individual" has an income greater than $50,000 a year.

Here is the set of variables contained in the data.

- age – The age of the individual
- type_employer – The type of employer the individual has. Whether they are government, military, private, an d so on.

- fnlwgt – The number of people the census takers believe that observation represents. We will be ignoring this variable
- education – The highest level of education achieved for that individual
- education_num – Highest level of education in numerical form
- marital – Marital status of the individual
- occupation – The occupation of the individual
- relationship – A bit more difficult to explain. Contains family relationship values like husband, father, and so on, but only contains one per observation. I'm not sure what this is supposed to represent
- race – descriptions of the individuals race. Black, White, Eskimo, and so on
- sex – Biological Sex
- capital_gain – Capital gains recorded
- capital_loss – Capital Losses recorded
- hr_per_week – Hours worked per week
- country – Country of origin for person
- income – Boolean Variable. Whether or not the person makes more than $50,000 per annum income.

8.6 References

- Trees with the rpart package
- Wholesale customers Data Set Origin of the data set of first example.
- Titanic: Getting Started With R - Part 3: Decision Trees. First understanding on how to read the graph of a tree.

- Classification and Regression Trees (CART) with rpart and rpart.plot. Got the `Titanic` example from there as well as a first understanding on pruning.

- Statistical Consulting Group. We learn here how to use the ROC curve. And we got out of it the `adult`dataset.
- A Complete Tutorial on Tree Based Modeling from Scratch (in R & Python). This website is a real gems as always.
- Stephen Milborrow. rpart.plot: Plot rpart Models. An Enhanced Version of plot.rpart., 2016. R Package. It is important to cite the very generous people who dedicates so much of their time to offer us great tool.

Chapter 9

Model Evaluation

9.1 Biais variance tradeoff

9.2 Bagging

9.3 Cross Validation

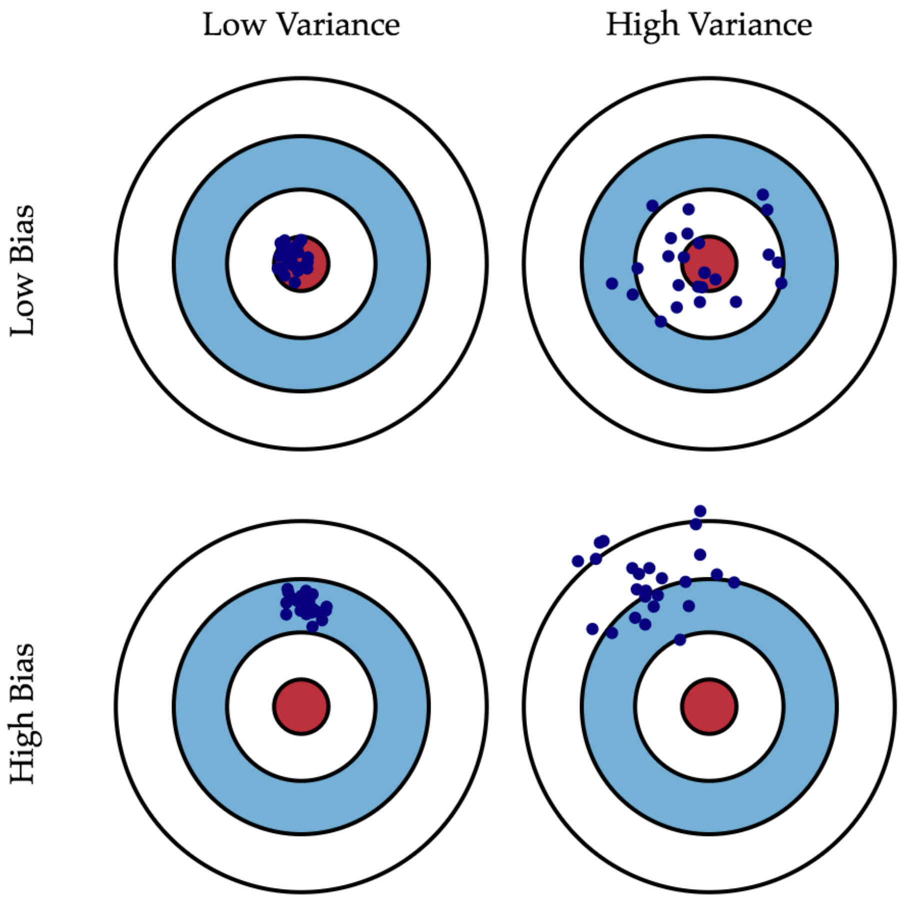


Figure 9.1: Biais Variance tradeoff

Chapter 10

Case Study - Predicting Survivalship on the Titanic

This chapter demonstrates another example of classification with machine learning. Kaggle made this exercise quite popular.

In this study, the training and test sets have already been defined, so we

10.1 Import the data.

We have put our data into our google drive here and here. You can find them on Kaggle if need be.

```r
library(tidyverse)

train_set <- read_csv("dataset/Kaggle_Titanic_train.csv")
test_set <- read_csv("dataset/Kaggle_Titanic_test.csv")

## Let's bind both set of data for our exploratory analysis.
df2 <- bind_rows(train_set, test_set)

## Let's have a first glimpse to our data
glimpse(df2)
```

```
## Observations: 1,309
## Variables: 12
## $ PassengerId <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15,...
## $ Survived    <int> 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, 1, 0,...
## $ Pclass      <int> 3, 1, 3, 1, 3, 3, 1, 3, 3, 2, 3, 1, 3, 3, 3, 2, 3,...
## $ Name        <chr> "Braund, Mr. Owen Harris", "Cumings, Mrs. John Bra...
## $ Sex         <chr> "male", "female", "female", "female", "male", "mal...
## $ Age         <dbl> 22, 38, 26, 35, 35, NA, 54, 2, 27, 14, 4, 58, 20, ...
## $ SibSp       <int> 1, 1, 0, 1, 0, 0, 0, 3, 0, 1, 1, 0, 0, 1, 0, 0, 4,...
## $ Parch       <int> 0, 0, 0, 0, 0, 0, 0, 1, 2, 0, 1, 0, 0, 5, 0, 0, 1,...
## $ Ticket      <chr> "A/5 21171", "PC 17599", "STON/O2. 3101282", "1138...
## $ Fare        <dbl> 7.2500, 71.2833, 7.9250, 53.1000, 8.0500, 8.4583, ...
## $ Cabin       <chr> NA, "C85", NA, "C123", NA, NA, "E46", NA, NA, NA, ...
## $ Embarked    <chr> "S", "C", "S", "S", "S", "Q", "S", "S", "S", "C", ...
```

10.2 Tidy the data

One can already see that we should put `Survived`, `Sex` and `Embarked` as factor.

```
df2$Survived <- factor(df2$Survived)
df2$Sex <- factor(df2$Sex)
df2$Embarked <- factor(df2$Embarked)
```

10.3 Understand the data

This step consists in massaging our variables to see if we can construct new ones or create additional meaning from what we have. This step require some additional knowledge related to the data and getting familiar with the topics at hand.

10.3.1 A. Transform the data

The great thing about this data set is all the features engineering one can do to increase the predictibilty power of our model.

10.3.1.1 Dealing with names.

One of the thing one can notice is the title associated with the name. The full names on their own might have little predictibility power, but the *title* in the name might have some value and can be used as an additional variables.

```
glimpse(df2$Name)
```

```
##  chr [1:1309] "Braund, Mr. Owen Harris" ...
```

```
## gsub is never fun to use.  But we need to strip the cell up to the comma,
## then everything after the point of the title.
df2$title <- gsub('(.*,)|(\\..*)', "", df2$Name)
table(df2$Sex,df2$title)
```

```
##
##          Capt  Col  Don  Dona  Dr  Jonkheer  Lady  Major  Master  Miss
##   female    0    0    0     1   1         0     1      0       0   260
##   male      1    4    1     0   7         1     0      2      61     0
##
##          Mlle  Mme  Mr  Mrs  Ms  Rev  Sir  the Countess
##   female    2    1   0  197   2    0    0             1
##   male      0    0 757    0   0    8    1             0
```

Some titles are just translations from other languages. Let's regroup those. Some other titles aren't occuring often and would not justify to have a category on their own. We have regroup some titles under common category. There is some arbitraire in here.

```r
df2$title <- gsub("Mlle", "Miss", df2$title)
df2$title <- gsub("Mme", "Mrs", df2$title)
df2$title <- gsub("Ms", "Miss", df2$title)
df2$title <- gsub("Jonkheer", "Mr", df2$title)
df2$title <- gsub("Capt|Col|Major", "Army", df2$title)
df2$title <- gsub("Don|Dona|Lady|Sir|the Countess", "Nobility", df2$title)
df2$title <- gsub("Dr|Rev", "Others", df2$title)
df2$title <- factor(df2$title)
df2$title <- factor(df2$title,
                    levels(df2$title)[c(5, 3, 2, 4, 7, 1, 6)] )
table(df2$Sex, df2$title)
```

```
##
##          Mrs  Miss  Master  Mr  Others  Army  Nobility
##   female 198   264       0   0       1     0         3
##   male     0     0      61 758      15     7         2
```

It would be also interesting in fact to check the proportion of survivors for each type of title.

```r
round(prop.table(table(df2$Survived, df2$title), 2), 2)
```

```
##
##      Mrs  Miss  Master   Mr  Others  Army  Nobility
##   0 0.21  0.30    0.42 0.84    0.77  0.60      0.25
##   1 0.79  0.70    0.57 0.16    0.23  0.40      0.75
```

We can notice that `Mrs` are more likely to survive than `Miss`. As expected, our `Mr` have a very low likelyhood of success. Our `Noble` title managed mostly to survive.

Our next step is to create a `Last_Name` variable. This could be helpful as the ways family have escaped the boat might hold some pattens.

```r
## To get the last name we strip everything after the first comma.
df2$last_name <- gsub(",.*", "", df2$Name)

## We can now put this as factor and check how many families.
```

```r
df2$last_name <- factor(df2$last_name)
```

So we have 875 different families on board of the Titanic. Of course, there might have different families with the same last name. If that's the case, we won't know.

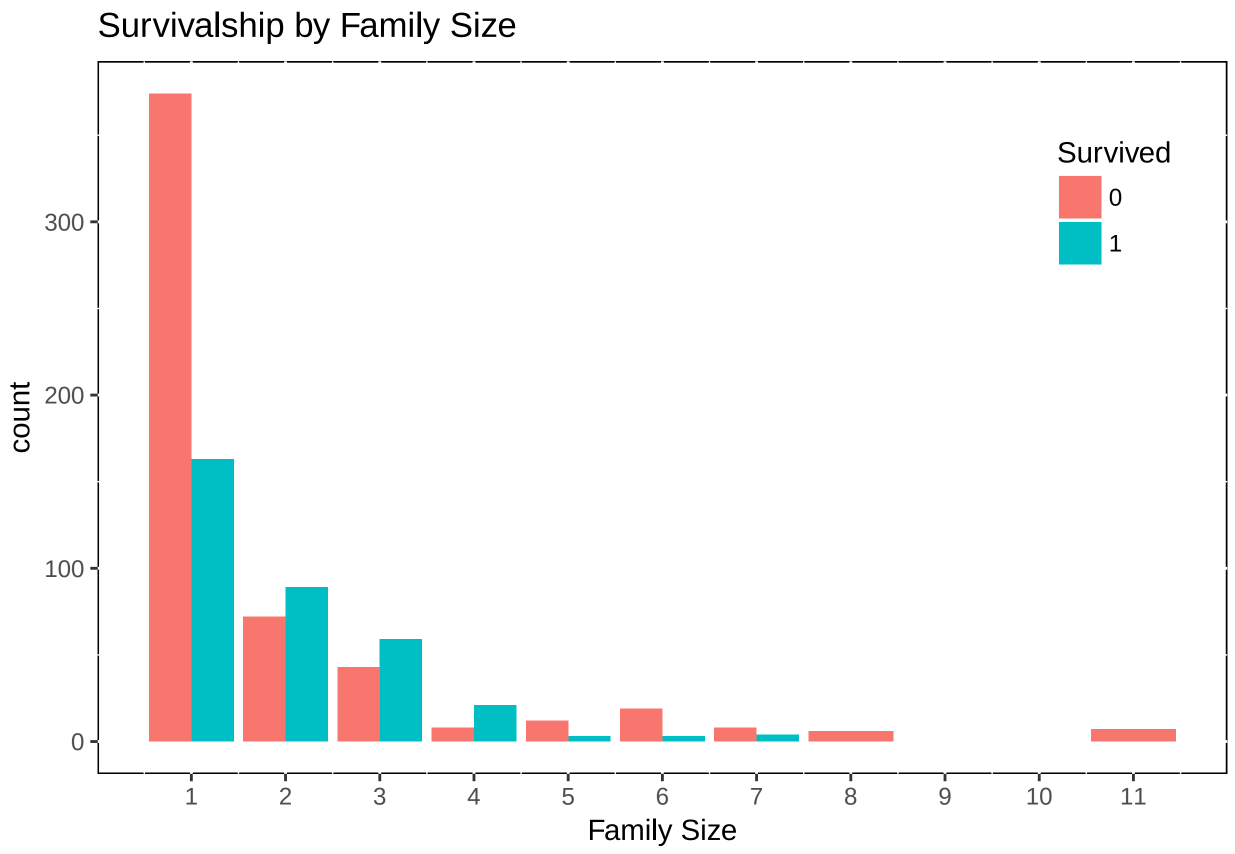
10.3.2 A. Vizualize with families.

We could add a variable about the family size.

```r
df2$family_size <- df2$SibSp + df2$Parch + 1
```

If we plot that to check survivalship in function of family size, one can notice interesting patterns.

```r
x <- df2[1:891,]
ggplot(x, aes(x = family_size, fill = factor(Survived))) +
  geom_bar(stat = 'count', position = "dodge") +
  scale_x_continuous(breaks = c(1:11)) +
  labs(x = "Family Size", fill = "Survived",
       title = "Survivalship by Family Size") +
  theme(legend.position = c(0.9, 0.8), panel.background = NULL)
```



Obviously, we only have the survivalship for the train set of data, as we have to guess the test set of data. So from what we have, there is a clear advantage in being a family of 2, 3 or 4. We could collapse the variable `Family_Size` into 3 levels.

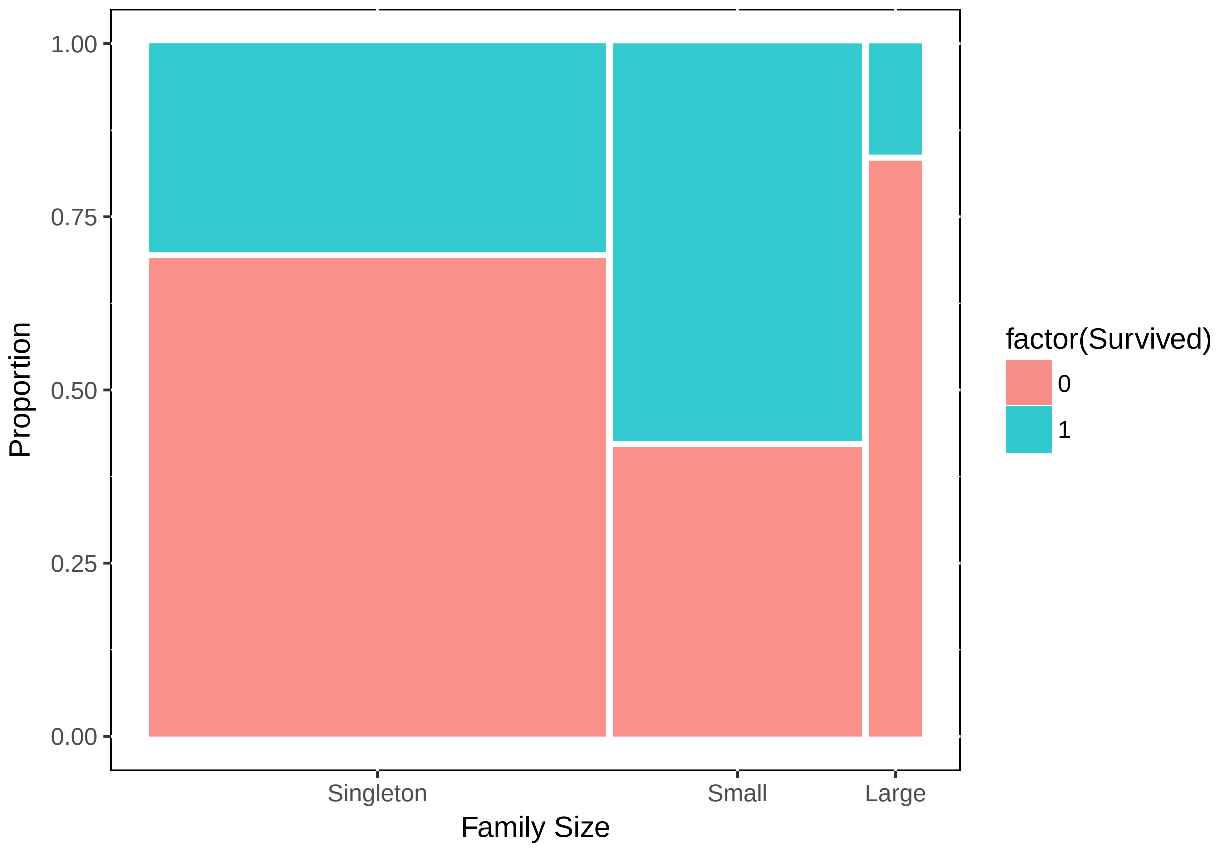
```
df2$family_size_type[df2$family_size == 1] <- "Singleton"
df2$family_size_type[df2$family_size <= 4 & df2$family_size > 1] <- "Small"
df2$family_size_type[df2$family_size > 4] <- "Large"
df2$family_size_type <- factor(df2$family_size_type, levels = c("Singleton", "Small", "L
```

We can see how many people in each category, then we plot the proportion of survivers in each category.

```
df3 <- df2[1:891,]
table(df3$Survived, df3$family_size_type)
```

```
##
##     Singleton Small Large
##   0       374   123    52
##   1       163   169    10
```

```
df3 <- as_tibble(df3)

library(ggmosaic)
ggplot(data = df3) +
  geom_mosaic(aes(weight = 1, x = product(family_size_type),
                  fill = factor(Survived), na.rm = TRUE)) +
  labs(x = "Family Size", y = "Proportion") +
  theme(panel.background = NULL)
```



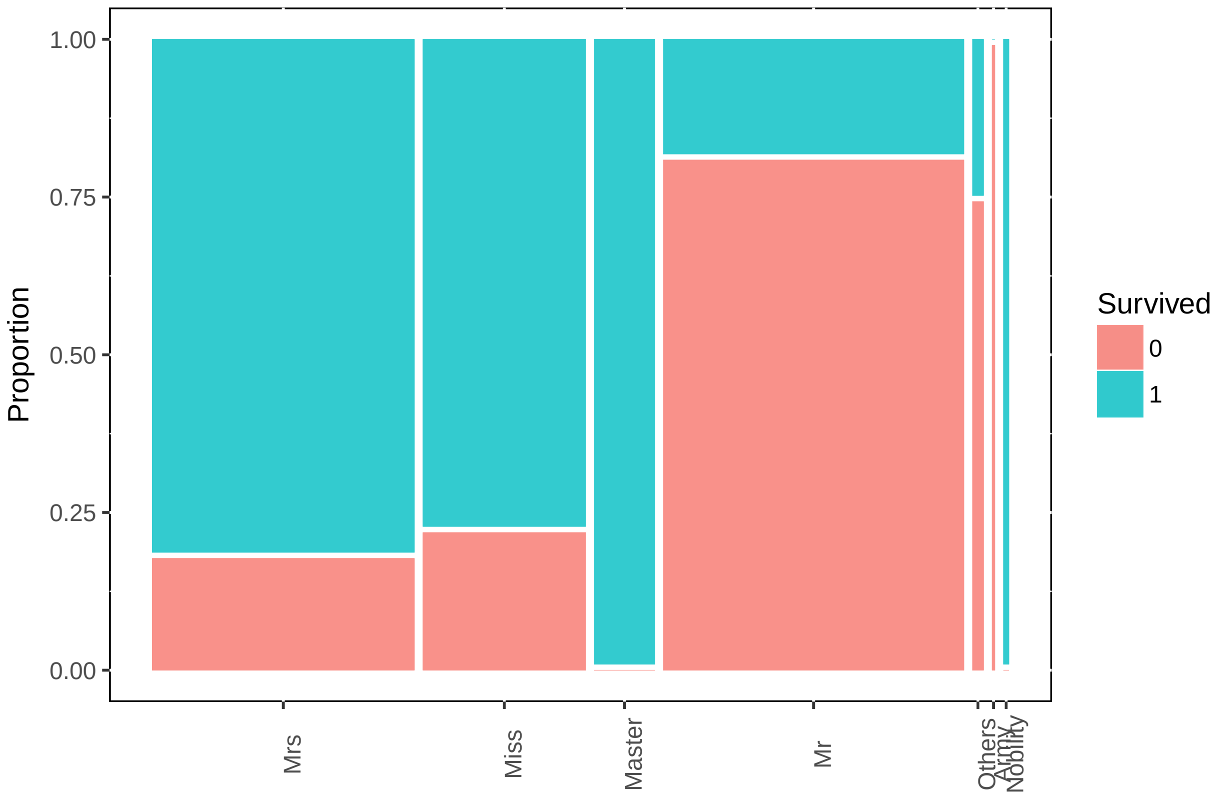
Clearly, there is an advantage in being in a family of size 2, 3 or 4; while there is a disadvantage in being part of of a bigger family.

We can try to digg in a bit further with our new family size and titles. For people who are part of a *Small* family size, which *title* are more likely to surived?

```
df4 <- df3 %>% dplyr::filter(family_size_type == "Small")
table(df4$Survived, df4$title)
```

```
##
##       Mrs  Miss  Master  Mr  Others  Army  Nobility
##    0   17    13       0  89       3     1         0
##    1   78    46      22  20       1     0         2
```

```
ggplot(data = df4) +
  geom_mosaic(aes(x = product(title), fill = Survived)) +
  labs(x = "Survivorship for Small Families in function of their title",
       y = "Proportion") +
  theme(panel.background = NULL, axis.text.x = element_text(angle=90, vjust=1))
```



Survivorship for Small Families in function of their title

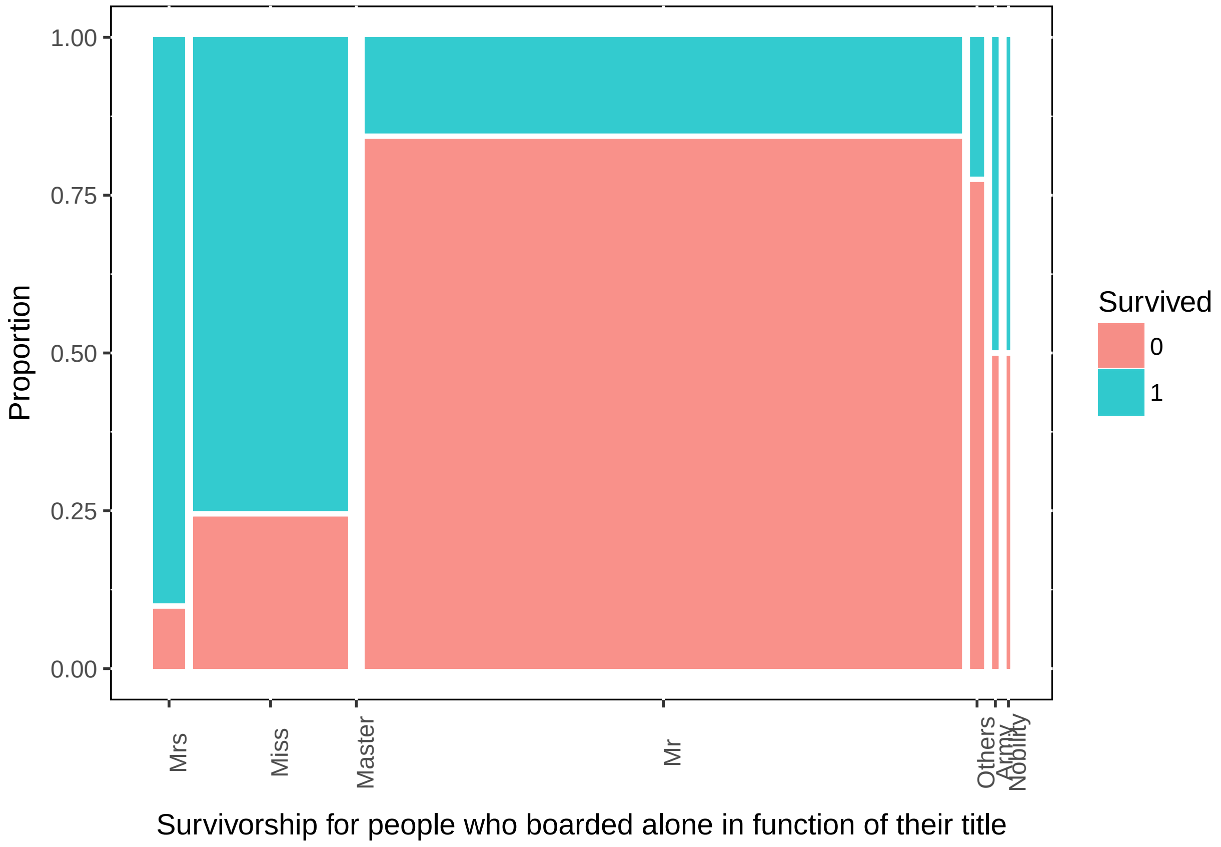
All masters in small families have survived. Miss & Mrs in small family size have also lots of chane of survival.

Similarly, for people who embarked alone (*Singleton*), which *title* are more likely to surived?

```
df4 <- df3 %>% filter(family_size_type == "Singleton")
table(df4$Survived, df4$title)
```

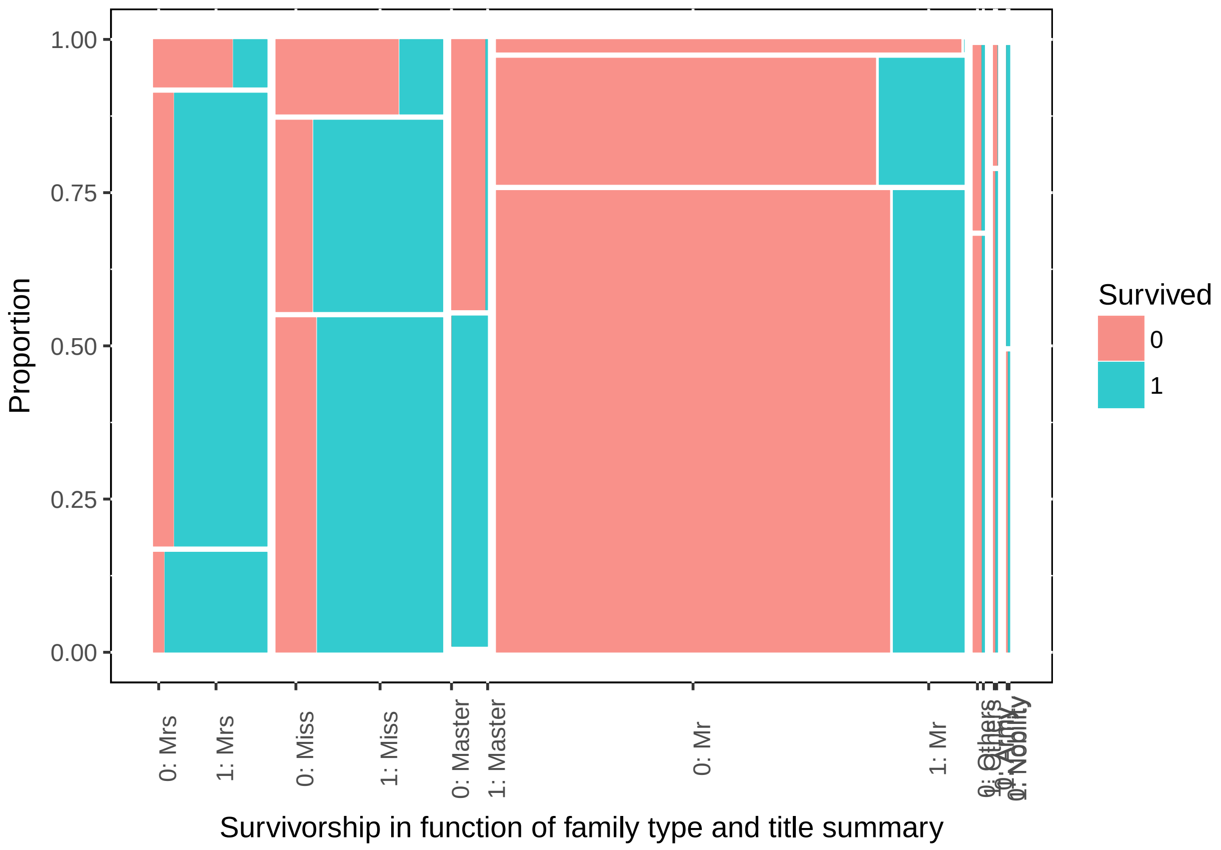
```
##
##      Mrs  Miss  Master  Mr  Others  Army  Nobility
##   0    2    25       0 337       7     2         1
##   1   19    78       0  61       2     2         1
```

```
ggplot(data = df4) + geom_mosaic(aes(x = product(title), fill = Survived)) +
  labs(x = "Survivorship for people who boarded alone in function of their title",
       y = "Proportion") +
  theme(panel.background = NULL, axis.text.x = element_text(angle=90, vjust=1))
```



It might not comes as clear, but we could do the same for title and gender. Vertically the stacks are ordered as `Singleton` then `Small` then `Large`.

```
ggplot(data = df3) + geom_mosaic(aes(x = product(family_size_type, title), fill = Survi
  labs(x = "Survivorship in function of family type and title summary",
       y = "Proportion") +
  theme(panel.background = NULL, axis.text.x = element_text(angle=90, vjust=1))
```



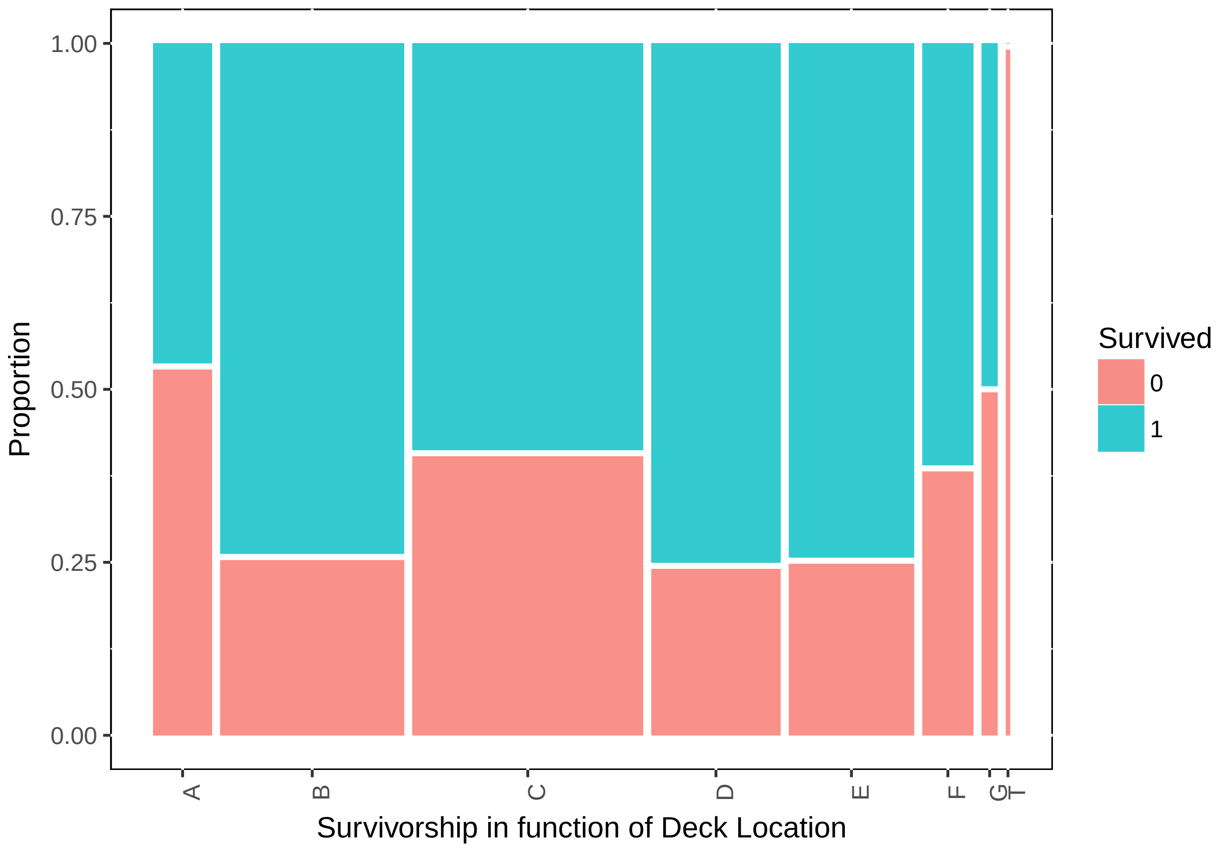
10.4 A. Visualize with cabins.

Although there are many missing data there, we can use the cabin number given to passengers. The first letter of the cabin number correspond to the deck on the boat. So let's strip that deck location from the cabin number.

```
df3$deck <- gsub("([A-Z]+).*", "\\1", df3$Cabin)
df4 <- df3 %>% filter(!is.na(deck))

table(df3$Survived, df3$deck)
```

```
##
##      A  B  C  D  E  F  G  T
##   0  8 12 24  8  8  5  2  1
##   1  7 35 35 25 24  8  2  0
```

```
ggplot(data = df4) + geom_mosaic(aes(x = product(deck), fill = Survived)) +
  labs(x = "Survivorship in function of Deck Location", y = "Proportion") +
  theme(panel.background = NULL, axis.text.x = element_text(angle=90, vjust=1))
```



```r
detach("package:ggmosaic", unload=TRUE)
```

There is a bit of an anomaly here as it almost as if most people survived. Now let's keep in mind, that this is only for people which we have their cabin data.

Let's have a look at how the `Passenger Class` are distributed on the decks. As we are also finishing this first round of feature engineering, let's just mention also how the Passenger Class is affecting survivalship.

```r
table(df3$Pclass, df3$deck)
```

```
##
##      A  B  C  D  E  F  G  T
##   1 15 47 59 29 25  0  0  1
##   2  0  0  0  4  4  8  0  0
##   3  0  0  0  0  3  5  4  0
```

```r
round(prop.table(table(df3$Survived, df3$Pclass), 2), 2)
```

```
##
##        1    2    3
##   0 0.37 0.53 0.76
##   1 0.63 0.47 0.24
```

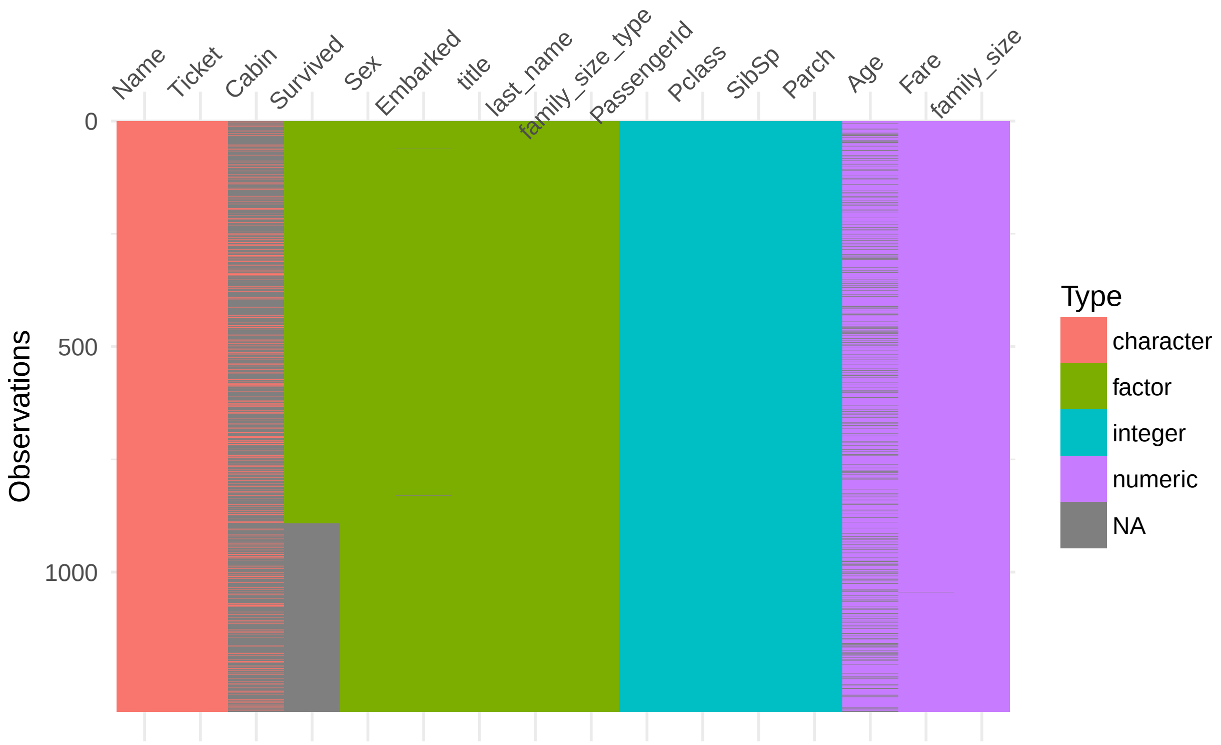
More first class people have survived than other classes.

10.5 B. Transform Dealing with missing data.

10.5.1 Overview.

I found this very cool package called `visdat` based on `ggplot2` that help us visualize easily missing data.

```
visdat::vis_dat(df2)
```



Straight away one can see that the variables `cabin` and and `Age` have quite a lot of missing data.
For more accuracy one could check

```
fun1 <- function(x){sum(is.na(x))}
map_dbl(df2, fun1)
```

```
##      PassengerId         Survived           Pclass             Name
##                0              418                0                0
##              Sex              Age            SibSp            Parch
##                0              263                0                0
##           Ticket             Fare            Cabin         Embarked
##                0                1             1014                2
##            title        last_name      family_size family_size_type
##                0                0                0                0
```

So we can see some missing data in `Fare` and in `Embarked` as well.
Let's deal with these last 2 variables first.

10.5.1.1 Basic Replacement.

We first start with the dessert and the variables that have few missing data. For those, one can take the median of similar data.

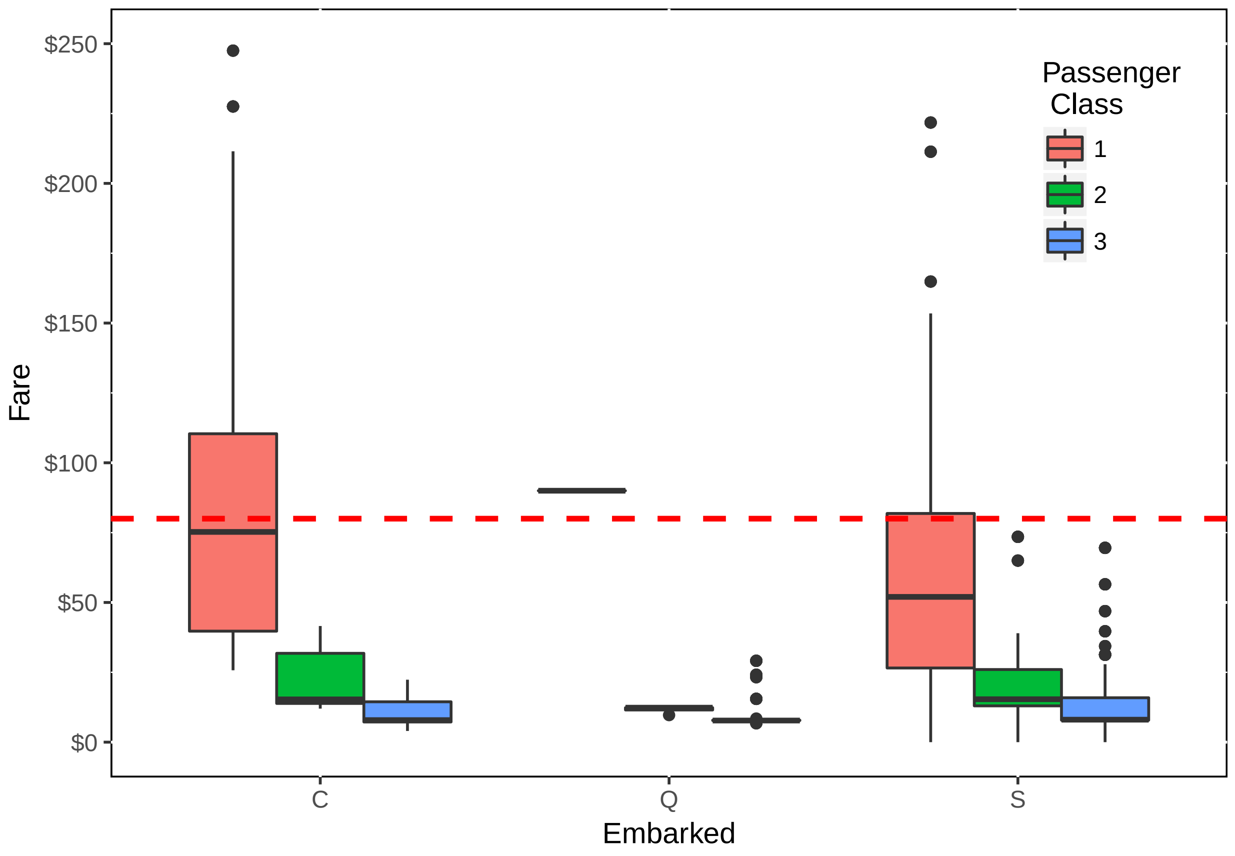
```
y <- which(is.na(df2$Embarked))
glimpse(df2[y, ])
```

```
## Observations: 2
## Variables: 16
## $ PassengerId      <int> 62, 830
## $ Survived         <fctr> 1, 1
## $ Pclass           <int> 1, 1
## $ Name             <chr> "Icard, Miss. Amelie", "Stone, Mrs. George Ne...
## $ Sex              <fctr> female, female
## $ Age              <dbl> 38, 62
## $ SibSp            <int> 0, 0
## $ Parch            <int> 0, 0
## $ Ticket           <chr> "113572", "113572"
## $ Fare             <dbl> 80, 80
## $ Cabin            <chr> "B28", "B28"
## $ Embarked         <fctr> NA, NA
## $ title            <fctr>  Miss,  Mrs
## $ last_name        <fctr> Icard, Stone
## $ family_size      <dbl> 1, 1
## $ family_size_type <fctr> Singleton, Singleton
```

So the 2 passengers that have no data on the origin of their embarqument are 2 ladies that boarded alone and that shared the same room in first class and that paid $80.

Let's see who might have paid $80 for a fare.

```
y <- df2 %>% filter(!is.na(Embarked))
ggplot(y, aes(x = Embarked, y = Fare, fill = factor(Pclass))) +
  geom_boxplot() +
  scale_y_continuous(labels = scales::dollar, limits = c(0, 250)) +
  labs(fill = "Passenger \n Class") +
  geom_hline(aes(yintercept = 80), color = "red", linetype = "dashed", lwd = 1) +
  theme(legend.position = c(0.9, 0.8), panel.background = NULL)
```



Following this graph, the 2 passengers without origin of embarcation are most likely from "C". That said, one can argue that the 2 ladies should have embarked from "S" as this is where most people embarked as shown in this table.

```r
table(df2$Embarked)
```

```
##
##   C   Q   S
## 270 123 914
```

That said, if we filter our data for the demographics of these 2 ladies, the likelihood of coming from "S" decreased quite a bit.

```r
x <- df2 %>% filter(Sex == "female", Pclass == 1, family_size == 1)
table(x$Embarked)
```

```
##
##  C  Q  S
## 30  0 20
```

So if we go with median price and with the demographics of the ladies, it would be more likely that they come from "C". So let's input that.

```r
df2$Embarked[c(62, 830)] <- "C"
```

Now onto that missing `Fare` data

```
y <- which(is.na(df2$Fare))
glimpse(df2[y, ])
```

```
## Observations: 1
## Variables: 16
## $ PassengerId      <int> 1044
## $ Survived         <fctr> NA
## $ Pclass           <int> 3
## $ Name             <chr> "Storey, Mr. Thomas"
## $ Sex              <fctr> male
## $ Age              <dbl> 60.5
## $ SibSp            <int> 0
## $ Parch            <int> 0
## $ Ticket           <chr> "3701"
## $ Fare             <dbl> NA
## $ Cabin            <chr> NA
## $ Embarked         <fctr> S
## $ title            <fctr>  Mr
## $ last_name        <fctr> Storey
## $ family_size      <dbl> 1
## $ family_size_type <fctr> Singleton
```

That passenger is a male that boarded in Southampton in third class. So let's take the median price for similar passagers.

```
y <- df2 %>% filter(Embarked == "S" & Pclass == "3" & Sex == "male" &
                        family_size == 1 & Age > 40)
median(y$Fare, na.rm = TRUE)
```

```
## [1] 7.8521
```

```
df2$Fare[1044] <- median(y$Fare, na.rm = TRUE)
```

10.5.1.2 Predictive modeling replacement.

First, we'll focus on the `Age` variable.
There are several methods to input missing data. We'll try 2 different ones in here.
But before we can go forward, we have to factorise some variables.
Let's do the same with `Sibsp` and `Parch`

```
df2$Pclass <- factor(df2$Pclass)
```

The first method we'll be using is with the `missForest` package.

```
y <- df2 %>% select(Pclass, Sex, Fare, Embarked, title, family_size, SibSp, Parch, Age)
y <- data.frame(y)
```

```
library(missForest)
z1 <- missForest(y, maxiter = 50, ntree = 500)
z1 <- z1[[1]]

# To view the new ages
# View(z1[[1]])

detach("package:missForest", unload=TRUE)
```

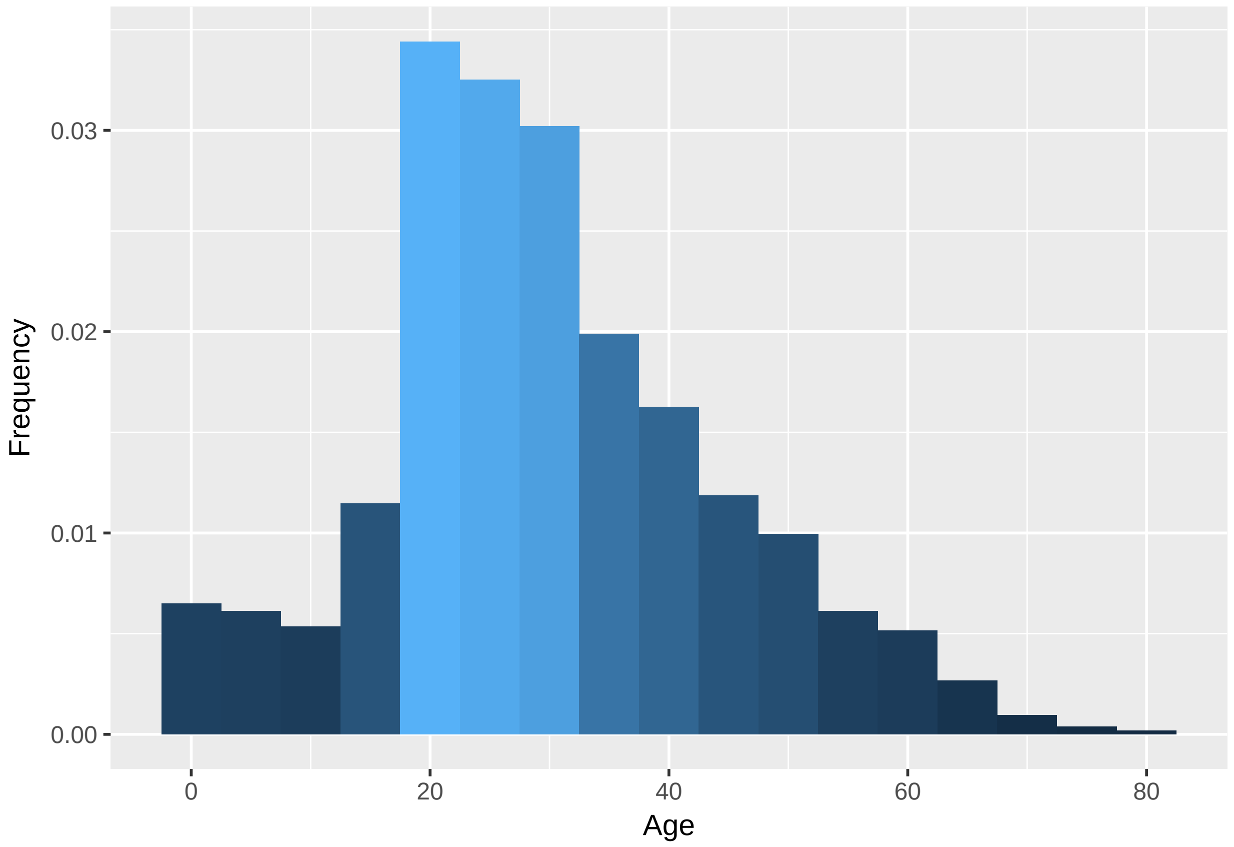
The process is fairly rapid on my computer (around 10~15 seconds)

Our second method takes slightly more time.
This time we are using the `mice` package.

```
y <- df2 %>% select(Pclass, Sex, Fare, Embarked, title, family_size, SibSp, Parch, Age)
y$Pclass <- factor(y$Pclass)
y$family_size <- factor(y$family_size)
y <- data.frame(y)

library(mice)
mice_mod <- mice(y, method = 'rf')
z2 <- complete(mice_mod)

# To view the new ages
#View(z2[[1]])

detach("package:mice", unload=TRUE)
```

let's compare both type of imputations.

```
p1 <- ggplot(df2, aes(x = df2$Age)) +
        geom_histogram(aes(y = ..density.., fill = ..count..),binwidth = 5) +
        labs(x = "Age", y = "Frequency", fil = "Survived") +
        theme(legend.position = "none")
p1
```



```r
p2 <- ggplot(z1, aes(x = z1$Age)) +
        geom_histogram(aes(y = ..density.., fill = ..count..),binwidth = 5) +
        labs(x = "Age", y = "Frequency", fil = "Survived") +
        theme(legend.position = "none")

p3 <- ggplot(z2, aes(x = z2$Age)) +
        geom_histogram(aes(y = ..density.., fill = ..count..),binwidth = 5) +
        labs(x = "Age", y = "Frequency", fil = "Survived") +
        theme(legend.position = "none")

multiplot(p1, p2, p3, cols = 3)
```

It does seem like our second method for imputation follow better our first graph. So let's use that one and input our predicted age into our main dataframe.

```r
# df2$Age <- z2$Age
```

10.5.2 C. Transform More feature engineering with the ages and others.

Now that we have filled the `NA` for the age variable. we can massage a bit more that variable. We can create 3 more variables: Infant from 0 to 5 years old. Child from 5 to 15 years old. Mothers if it is a woman with the variable `Parch` which is greater than one.

```
df2$infant <- factor(if_else(df2$Age <= 5, 1, 0))
df2$child <- factor(if_else((df2$Age > 5 & df2$Age < 15), 1, 0))
```

```
df2$mother <- factor(if_else((df2$Sex == "female" & df2$Parch != 0), 1, 0))
df2$single <- factor(if_else((df2$SibSp + df2$Parch + 1 == 1), 1, 0))
```

10.6 References.

- Exploring the titanic dataset from Megan Risdal. here
- The `visdat` package. here
- The `ggmosaic` package. here

Chapter 11

Case Study - Mushrooms Classification

This example demonstrates how to classify muhsrooms as edible or not. It also answer the question: what are the main characteristics of an edible mushroom?

This blog post gave us first the idea and we followed most of it. We also noticed that Kaggle has put online the same data set and classification exercise. We have taken inspiration from some posts here and here

The data set is available on the Machine Learning Repository of the UC Irvine website.

11.1 Import the data

The data set is given to us in a rough form and quite a bit of editing is necessary.

```r
# Load the data - we downloaded the data from the website and saved it into a .csv fil
library(tidyverse)
mushroom <- read_csv("dataset/Mushroom.csv", col_names = FALSE)
glimpse(mushroom)
```

```
## Observations: 8,124
## Variables: 23
## $ X1  <chr> "p", "e", "e", "p", "e", "e", "e", "e", "p", "e", "e", "e"...
## $ X2  <chr> "x", "x", "b", "x", "x", "x", "b", "b", "x", "b", "x", "x"...
## $ X3  <chr> "s", "s", "s", "y", "s", "y", "s", "y", "y", "s", "y", "y"...
## $ X4  <chr> "n", "y", "w", "w", "g", "y", "w", "w", "w", "y", "y", "y"...
## $ X5  <chr> "t", "t", "t", "t", "f", "t", "t", "t", "t", "t", "t", "t"...
## $ X6  <chr> "p", "a", "l", "p", "n", "a", "a", "l", "p", "a", "l", "a"...
## $ X7  <chr> "f", "f", "f", "f", "f", "f", "f", "f", "f", "f", "f", "f"...
## $ X8  <chr> "c", "c", "c", "c", "w", "c", "c", "c", "c", "c", "c", "c"...
## $ X9  <chr> "n", "b", "b", "n", "b", "b", "b", "b", "n", "b", "b", "b"...
## $ X10 <chr> "k", "k", "n", "n", "k", "n", "g", "n", "p", "g", "g", "n"...
## $ X11 <chr> "e", "e", "e", "e", "t", "e", "e", "e", "e", "e", "e", "e"...
## $ X12 <chr> "e", "c", "c", "e", "e", "c", "c", "c", "e", "c", "c", "c"...
## $ X13 <chr> "s", "s", "s", "s", "s", "s", "s", "s", "s", "s", "s", "s"...
## $ X14 <chr> "s", "s", "s", "s", "s", "s", "s", "s", "s", "s", "s", "s"...
## $ X15 <chr> "w", "w", "w", "w", "w", "w", "w", "w", "w", "w", "w", "w"...
## $ X16 <chr> "w", "w", "w", "w", "w", "w", "w", "w", "w", "w", "w", "w"...
## $ X17 <chr> "p", "p", "p", "p", "p", "p", "p", "p", "p", "p", "p", "p"...
## $ X18 <chr> "w", "w", "w", "w", "w", "w", "w", "w", "w", "w", "w", "w"...
## $ X19 <chr> "o", "o", "o", "o", "o", "o", "o", "o", "o", "o", "o", "o"...
## $ X20 <chr> "p", "p", "p", "p", "e", "p", "p", "p", "p", "p", "p", "p"...
## $ X21 <chr> "k", "n", "n", "k", "n", "k", "k", "n", "k", "k", "n", "k"...
## $ X22 <chr> "s", "n", "n", "s", "a", "n", "n", "s", "v", "s", "n", "s"...
## $ X23 <chr> "u", "g", "m", "u", "g", "g", "m", "m", "g", "m", "g", "m"...
```

Basically we have 8124 mushrooms in the dataset. And each observation consists of 23 variables. As it stands, the data frame doesn't look very meaningfull. We have to go back to the source to bring meaning to each of the variables and to the various levels of the categorical variables.

11.2 Tidy the data

This is the least fun part of the workflow.
We'll start by giving names to each of the variables, then we specify the category for each variable. It is not necessary to do so but it does add meaning to what we do.

```r
# Rename the variables
colnames(mushroom) <- c("edibility", "cap_shape", "cap_surface",
                        "cap_color", "bruises", "odor",
                        "gill_attachement", "gill_spacing", "gill_size",
                        "gill_color", "stalk_shape", "stalk_root",
                        "stalk_surface_above_ring", "stalk_surface_below_ring", "stalk_c
                        "stalk_color_below_ring", "veil_type", "veil_color",
                        "ring_number", "ring_type", "spore_print_color",
                        "population", "habitat")

# Defining the levels for the categorical variables
## We make each variable as a factor
mushroom <- mushroom %>% map_df(function(.x) as.factor(.x))

## We redefine each of the category for each of the variables
levels(mushroom$edibility) <- c("edible", "poisonous")
levels(mushroom$cap_shape) <- c("bell", "conical", "flat", "knobbed", "sunken", "convex'
levels(mushroom$cap_color) <- c("buff", "cinnamon", "red", "gray", "brown", "pink",
                                "green", "purple", "white", "yellow")
levels(mushroom$cap_surface) <- c("fibrous", "grooves", "scaly", "smooth")
levels(mushroom$bruises) <- c("no", "yes")
levels(mushroom$odor) <- c("almond", "creosote", "foul", "anise", "musty", "none", "pung
levels(mushroom$gill_attachement) <- c("attached", "free")
levels(mushroom$gill_spacing) <- c("close", "crowded")
levels(mushroom$gill_size) <- c("broad", "narrow")
levels(mushroom$gill_color) <- c("buff", "red", "gray", "chocolate", "black", "brown", "
                                 "pink", "green", "purple", "white", "yellow")
levels(mushroom$stalk_shape) <- c("enlarging", "tapering")
levels(mushroom$stalk_root) <- c("missing", "bulbous", "club", "equal", "rooted")
levels(mushroom$stalk_surface_above_ring) <- c("fibrous", "silky", "smooth", "scaly")
levels(mushroom$stalk_surface_below_ring) <- c("fibrous", "silky", "smooth", "scaly")
levels(mushroom$stalk_color_above_ring) <- c("buff", "cinnamon", "red", "gray", "brown",
                                "green", "purple", "white", "yellow")
levels(mushroom$stalk_color_below_ring) <- c("buff", "cinnamon", "red", "gray", "brown",
                                "green", "purple", "white", "yellow")
levels(mushroom$veil_type) <- "partial"
levels(mushroom$veil_color) <- c("brown", "orange", "white", "yellow")
levels(mushroom$ring_number) <- c("none", "one", "two")
levels(mushroom$ring_type) <- c("evanescent", "flaring", "large", "none", "pendant")
levels(mushroom$spore_print_color) <- c("buff", "chocolate", "black", "brown", "orange",
                                        "green", "purple", "white", "yellow")
levels(mushroom$population) <- c("abundant", "clustered", "numerous", "scattered", "seve
levels(mushroom$habitat) <- c("wood", "grasses", "leaves", "meadows", "paths", "urban",
```

Let's check our changes one last time before diving into in the next phase of our data analysis workflow.

```
glimpse(mushroom)
```

```
## Observations: 8,124
## Variables: 23
## $ edibility                <fctr> poisonous, edible, edible, poisonous...
## $ cap_shape                <fctr> convex, convex, bell, convex, convex...
## $ cap_surface              <fctr> scaly, scaly, scaly, smooth, scaly, ...
## $ cap_color                <fctr> brown, yellow, white, white, gray, y...
## $ bruises                  <fctr> yes, yes, yes, yes, no, yes, yes, ye...
## $ odor                     <fctr> pungent, almond, anise, pungent, non...
## $ gill_attachement         <fctr> free, free, free, free, free, free, ...
## $ gill_spacing             <fctr> close, close, close, close, crowded,...
## $ gill_size                <fctr> narrow, broad, broad, narrow, broad,...
## $ gill_color               <fctr> black, black, brown, brown, black, b...
## $ stalk_shape              <fctr> enlarging, enlarging, enlarging, enl...
## $ stalk_root               <fctr> equal, club, club, equal, equal, clu...
## $ stalk_surface_above_ring <fctr> smooth, smooth, smooth, smooth, smoo...
## $ stalk_surface_below_ring <fctr> smooth, smooth, smooth, smooth, smoo...
## $ stalk_color_above_ring   <fctr> purple, purple, purple, purple, purp...
## $ stalk_color_below_ring   <fctr> purple, purple, purple, purple, purp...
## $ veil_type                <fctr> partial, partial, partial, partial, ...
## $ veil_color               <fctr> white, white, white, white, white, w...
## $ ring_number              <fctr> one, one, one, one, one, one, one, o...
## $ ring_type                <fctr> pendant, pendant, pendant, pendant, ...
## $ spore_print_color        <fctr> black, brown, brown, black, brown, b...
## $ population               <fctr> scattered, numerous, numerous, scatt...
## $ habitat                  <fctr> urban, grasses, meadows, urban, gras...
```

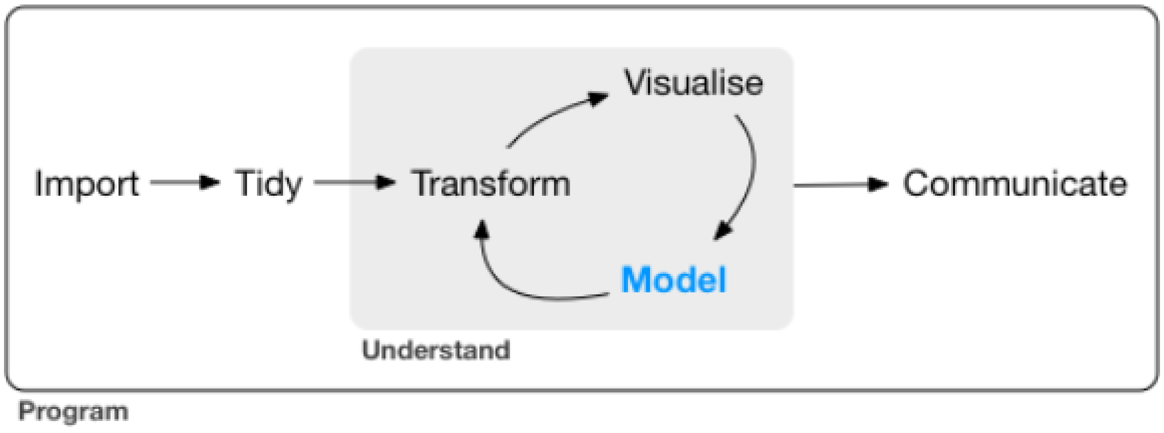
As each variables is categorical, let's see how many categories are we speaking about?

```r
number_class <- function(x){
  x <- length(levels(x))
}

x <- mushroom %>% map_dbl(function(.x) number_class(.x)) %>% as_tibble() %>%
       rownames_to_column() %>% arrange(desc(value))
colnames(x) <- c("Variable name", "Number of levels")
print(x)
```

```
## # A tibble: 23 x 2
##          `Variable name` `Number of levels`
##                    <chr>              <dbl>
##  1            gill_color                 12
##  2             cap_color                 10
##  3 stalk_color_above_ring                10
##  4 stalk_color_below_ring                10
##  5                  odor                  9
##  6     spore_print_color                  9
##  7               habitat                  7
##  8             cap_shape                  6
##  9            population                  6
## 10            stalk_root                  5
## # ... with 13 more rows
```

11.3 Understand the data

This is the circular phase of our dealing with data. This is where each of the transforming, visualizing and modeling stage reinforce each other to create a better understanding.



11.3.1 Transform the data

We noticed from the previous section an issue with the veil_type variable. It has only one factor. So basically, it does not bring any information. Furthermore, factor variable with only one level do create issues later on at the modeling stage. R will throw out an error for the categorical variable that has only one level.
So let's take away that column.

```
mushroom <- mushroom %>% select(- veil_type)
```

Do we have any missing data? Most ML algorithms won't work if we have missing data.

```
map_dbl(mushroom, function(.x) {sum(is.na(.x))})
```

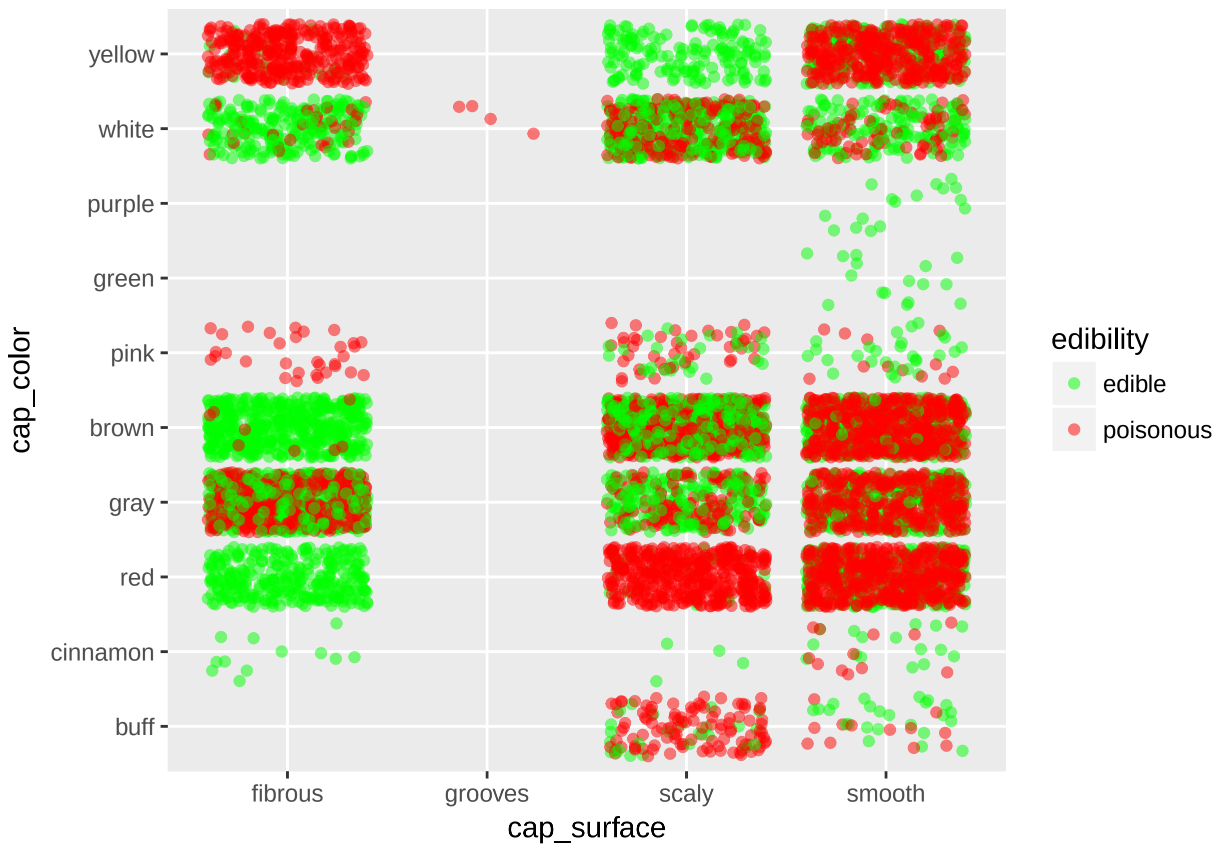
```
##                edibility                cap_shape               cap_surface
##                        0                        0                         0
##                cap_color                  bruises                      odor
##                        0                        0                         0
##          gill_attachement             gill_spacing                 gill_size
##                        0                        0                         0
##               gill_color              stalk_shape                stalk_root
##                        0                        0                         0
## stalk_surface_above_ring stalk_surface_below_ring    stalk_color_above_ring
##                        0                        0                         0
##    stalk_color_below_ring               veil_color               ring_number
##                        0                        0                         0
##                ring_type        spore_print_color                population
##                        0                        0                         0
##                  habitat
##                        0
```

Lucky us! We have no missing data.

11.3.2 Visualize the data

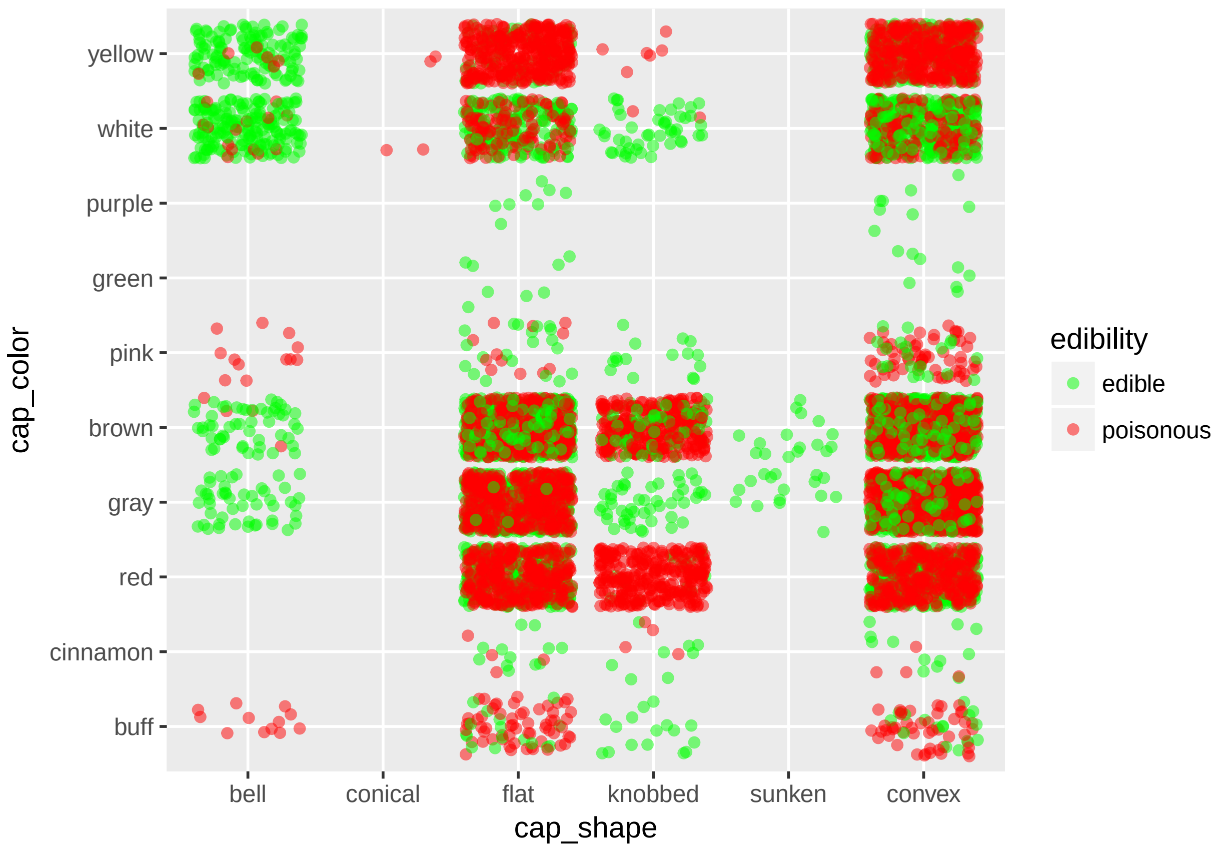
This is one of the most important step in the DS process. This stage can gives us unexpected insights and often allows us to ask the right questions.

```r
library(ggplot2)
ggplot(mushroom, aes(x = cap_surface, y = cap_color, col = edibility)) +
  geom_jitter(alpha = 0.5) +
  scale_color_manual(breaks = c("edible", "poisonous"),
                     values = c("green", "red"))
```



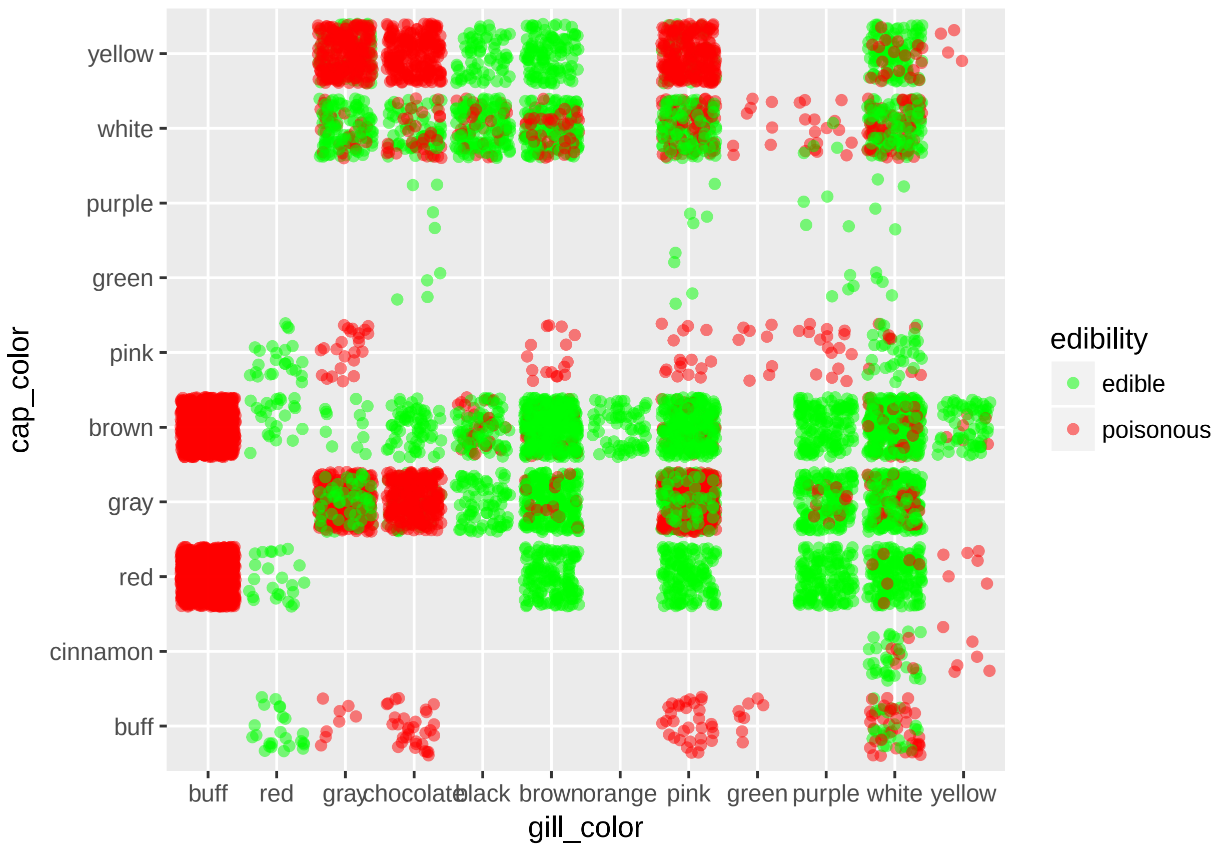
If we want to stay safe, better bet on *fibrous* surface. Stay especially away from *smooth* surface, except if they are purple or green.

```r
ggplot(mushroom, aes(x = cap_shape, y = cap_color, col = edibility)) +
  geom_jitter(alpha = 0.5) +
  scale_color_manual(breaks = c("edible", "poisonous"),
                     values = c("green", "red"))
```

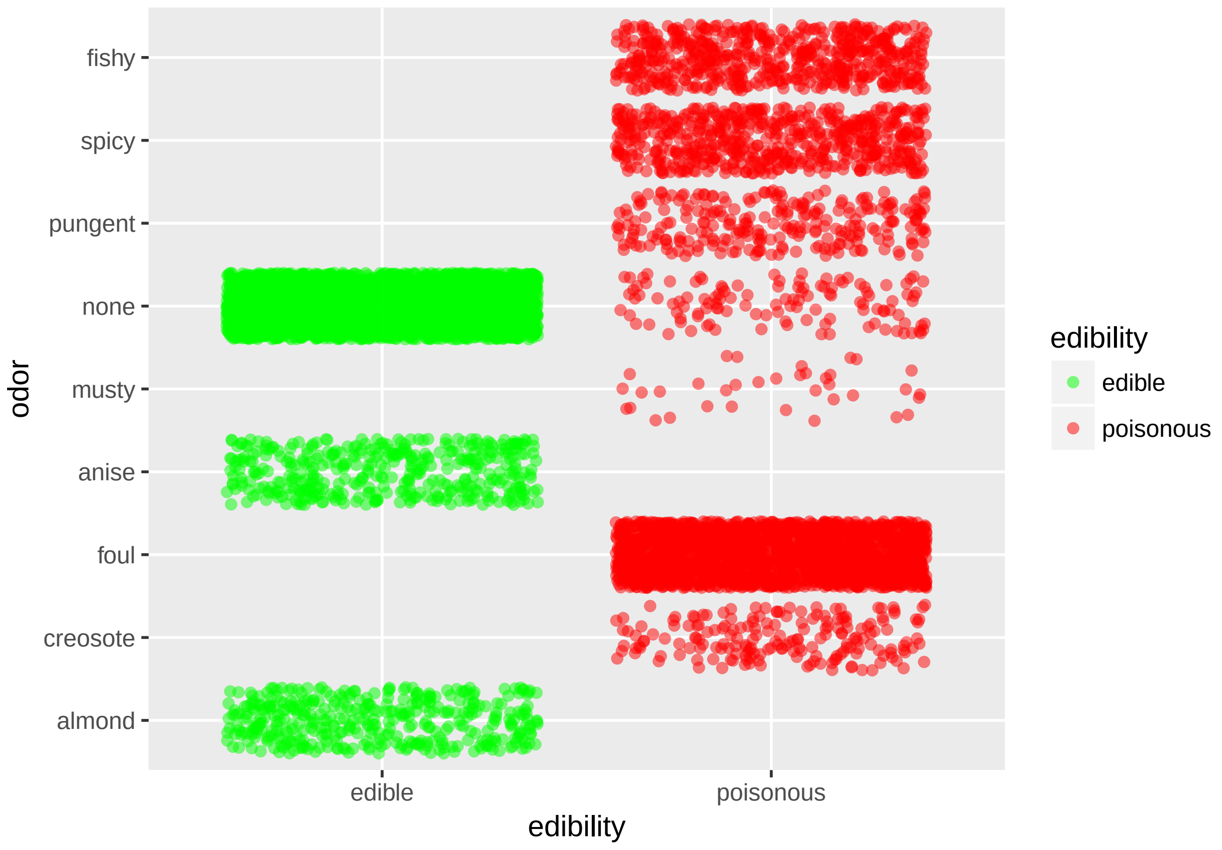


Again, in case one don't know about mushroom, it is better to stay away from all shapes except maybe for *bell* shape mushrooms.

```
ggplot(mushroom, aes(x = gill_color, y = cap_color, col = edibility)) +
  geom_jitter(alpha = 0.5) +
  scale_color_manual(breaks = c("edible", "poisonous"),
                     values = c("green", "red"))
```



```r
ggplot(mushroom, aes(x = edibility, y = odor, col = edibility)) +
  geom_jitter(alpha = 0.5) +
  scale_color_manual(breaks = c("edible", "poisonous"),
                     values = c("green", "red"))
```



Odor is defintely quite an informative predictor. Basically, if it smells *fishy*, *spicy* or *pungent* just stay away. If it smells like *anise* or *almond* you can go ahead. If it doesn't smell anything, you have better chance that it is edible than not.

TO DO: put a comment on what we see TO DO: put a mosaic graph

11.3.3 Modeling

At this stage, we should have gathered enough information and insights on our data to choose appropriate modeling techniques.

Before we go ahead, we need to split the data into a training and testing set

```
set.seed(1810)
mushsample <- caret::createDataPartition(y = mushroom$edibility, times = 1, p = 0.8, li
train_mushroom <- mushroom[mushsample, ]
test_mushroom <- mushroom[-mushsample, ]
```

We can check the quality of the splits in regards to our predicted (dependent) variable.

```
round(prop.table(table(mushroom$edibility)), 2)
```

```
##
##    edible poisonous
##      0.52      0.48
```

```r
round(prop.table(table(train_mushroom$edibility)), 2)
```

```
##
##    edible poisonous
##      0.52      0.48
```

```r
round(prop.table(table(test_mushroom$edibility)), 2)
```

```
##
##    edible poisonous
##      0.52      0.48
```

It seems like we have the right splits.

11.3.3.1 Use of Regression Tree

As we have many categorical variables, regression tree is an ideal classification tools for such situation.
We'll use the `rpart` package. Let's give it a try without any customization.

```r
library(rpart)
library(rpart.plot)
set.seed(1810)
model_tree <- rpart(edibility ~ ., data = train_mushroom, method = "class")
model_tree
```

```
## n= 6500
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
## 1) root 6500 3133 edible (0.51800000 0.48200000)
##   2) odor=almond,anise,none 3468  101 edible (0.97087659 0.02912341)
##    4) spore_print_color=buff,chocolate,black,brown,orange,purple,white,yellow 3408  41
##     5) spore_print_color=green 60    0 poisonous (0.00000000 1.00000000) *
##   3) odor=creosote,foul,musty,pungent,spicy,fishy 3032   0 poisonous (0.00000000 1.00000
```

```r
caret::confusionMatrix(data=predict(model_tree, type = "class"),
                       reference = train_mushroom$edibility,
                       positive="edible")
```

```
## Confusion Matrix and Statistics
##
##             Reference
## Prediction  edible poisonous
##   edible      3367        41
##   poisonous      0      3092
##
##                Accuracy : 0.9937
##                  95% CI : (0.9915, 0.9955)
##     No Information Rate : 0.518
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.9874
##  Mcnemar's Test P-Value : 4.185e-10
##
##             Sensitivity : 1.0000
##             Specificity : 0.9869
##          Pos Pred Value : 0.9880
##          Neg Pred Value : 1.0000
##              Prevalence : 0.5180
##          Detection Rate : 0.5180
##    Detection Prevalence : 0.5243
##       Balanced Accuracy : 0.9935
##
##        'Positive' Class : edible
##
```

We have quite an issue here. 40 mushrooms have been predicted as edible but were actually poisonous. That should not be happening. So we'll set up a penalty for wrongly predicting a mushroom as `edible` when in reality it is `poisonous`. A mistake the other way is not as bad. At worst we miss on a good recipe! So let's redo our tree with a penalty for wrongly predicting poisonous. To do this, we introduce a penalty matrix that we'll use as a parameter in our rpart function.

```
penalty_matrix <- matrix(c(0, 1, 10, 0), byrow = TRUE, nrow = 2)
model_tree_penalty <- rpart(edibility ~ ., data = train_mushroom, method = "class",
                    parms = list(loss = penalty_matrix))

caret::confusionMatrix(data=predict(model_tree_penalty, type = "class"),
                       reference = train_mushroom$edibility,
                       positive="edible")
```

```
## Confusion Matrix and Statistics
##
##            Reference
## Prediction  edible poisonous
##   edible      3367         0
##   poisonous      0      3133
##
##                Accuracy : 1
##                  95% CI : (0.9994, 1)
##     No Information Rate : 0.518
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 1
##  Mcnemar's Test P-Value : NA
##
##             Sensitivity : 1.000
##             Specificity : 1.000
##          Pos Pred Value : 1.000
##          Neg Pred Value : 1.000
##              Prevalence : 0.518
##          Detection Rate : 0.518
##    Detection Prevalence : 0.518
##       Balanced Accuracy : 1.000
##
##        'Positive' Class : edible
##
```

So introducing a penalty did the job; it gave us a perfect prediction and saves us from a jounrey at the hospital.

Another way to increase the accuracy of our tree model is to play on the `cp` parameter. We start to build a tree with a very low `cp` (that is we'll have a deep tree). The idea is that we then prune it later.

```r
model_tree <- rpart(edibility ~ ., data = train_mushroom,
                    method = "class", cp = 0.00001)
```

To prune a tree, we first have to find the `cp` that gives the lowest `xerror` or cross-validation error. We can find the lowest `xerror` using either the `printcp` or `plotcp` function.

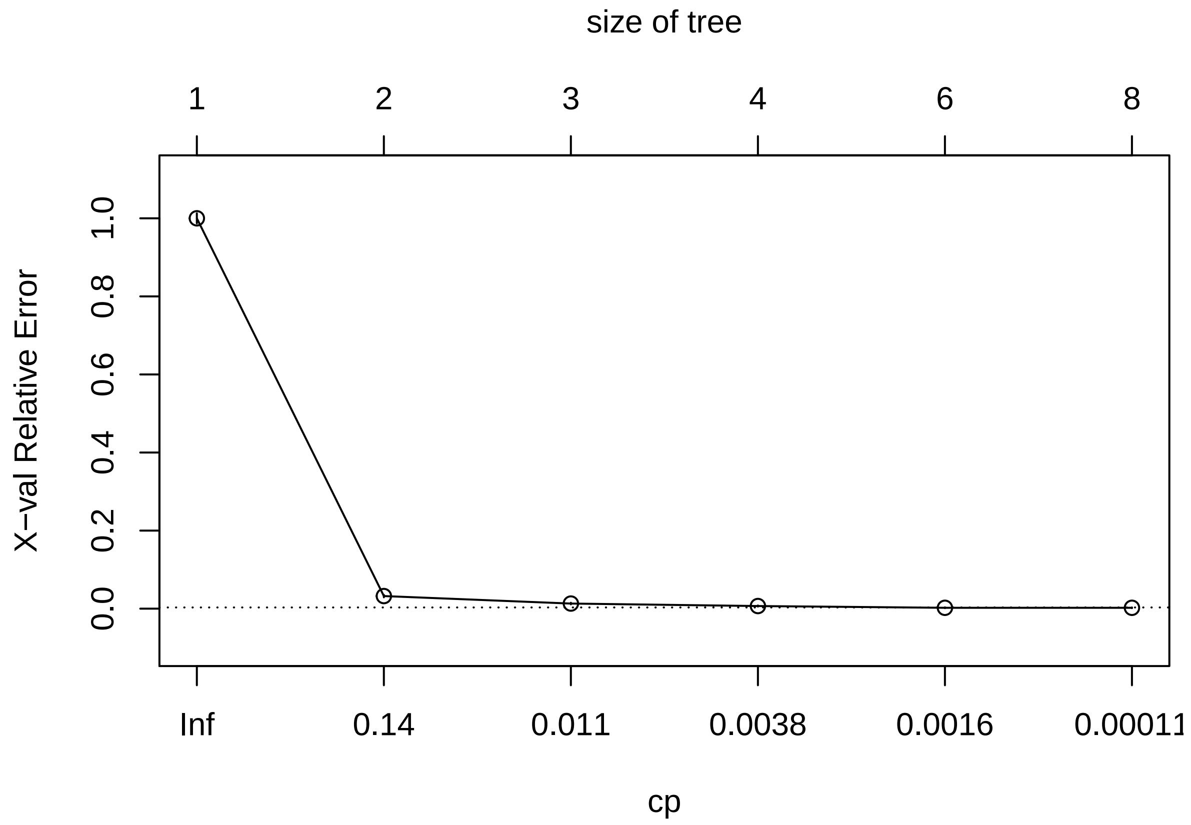
```r
printcp(model_tree)
```

```
##
## Classification tree:
## rpart(formula = edibility ~ ., data = train_mushroom, method = "class",
##     cp = 1e-05)
##
## Variables actually used in tree construction:
## [1] cap_surface            habitat                odor
## [4] spore_print_color      stalk_color_below_ring stalk_root
##
## Root node error: 3133/6500 = 0.482
##
## n= 6500
##
```

```
##          CP nsplit rel error    xerror       xstd
## 1 0.9677625      0 1.0000000 1.0000000 0.01285833
## 2 0.0191510      1 0.0322375 0.0322375 0.00318273
## 3 0.0063837      2 0.0130865 0.0130865 0.00203731
## 4 0.0022343      3 0.0067028 0.0067028 0.00146032
## 5 0.0011171      5 0.0022343 0.0022343 0.00084402
## 6 0.0000100      7 0.0000000 0.0022343 0.00084402
```

We can see here that that the lowest `xerror` happen at the 5th split.

```r
plotcp(model_tree)
```



```r
model_tree$cptable[which.min(model_tree$cptable[, "xerror"]), "CP"]
```

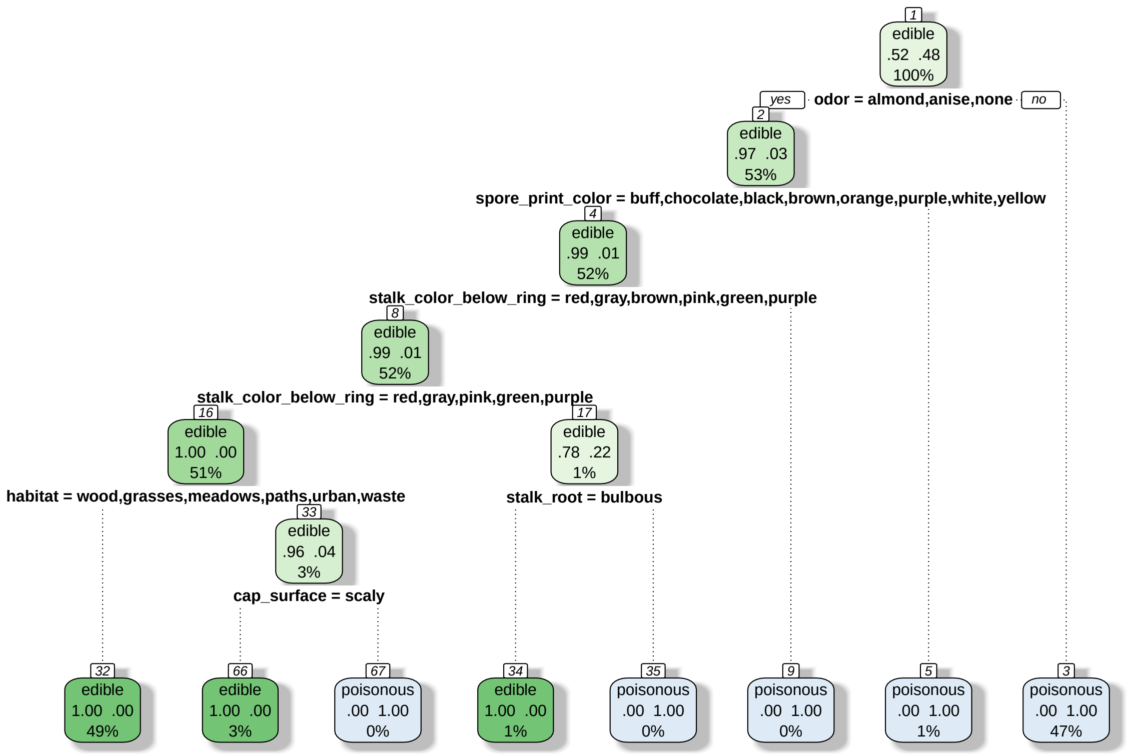
```
## [1] 0.00111714
```

So now we can start pruning our tree with the `cp` that gives the lowest cross-validation error.

```r
bestcp <- round(model_tree$cptable[which.min(model_tree$cptable[, "xerror"]), "CP"], 4)
model_tree_pruned <- prune(model_tree, cp = bestcp)
```

Let's have a quick look at the tree as it stands

```r
rpart.plot(model_tree_pruned, extra = 104, box.palette = "GnBu",
           branch.lty = 3, shadow.col = "gray", nn = TRUE)
```



How does the model perform on the train data?

```
#table(train_mushroom$edibility, predict(model_tree, type="class"))

caret::confusionMatrix(data=predict(model_tree_pruned, type = "class"),
                       reference = train_mushroom$edibility,
                       positive="edible")
```

```
## Confusion Matrix and Statistics
##
##             Reference
## Prediction   edible poisonous
##   edible       3367         0
##   poisonous       0      3133
##
##                Accuracy : 1
##                  95% CI : (0.9994, 1)
##     No Information Rate : 0.518
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 1
##  Mcnemar's Test P-Value : NA
##
##             Sensitivity : 1.000
##             Specificity : 1.000
##          Pos Pred Value : 1.000
##          Neg Pred Value : 1.000
##              Prevalence : 0.518
##          Detection Rate : 0.518
##    Detection Prevalence : 0.518
##       Balanced Accuracy : 1.000
##
##        'Positive' Class : edible
##
```

It seems like we have a perfect accuracy on our training set. It is quite rare to have such perfect accuracy.

Let's check how it fares on the testing set.

```
test_tree <- predict(model_tree, newdata = test_mushroom)
caret::confusionMatrix(data = predict(model_tree, newdata = test_mushroom, type = "clas
                       reference = test_mushroom$edibility,
                       positive = "edible")
```

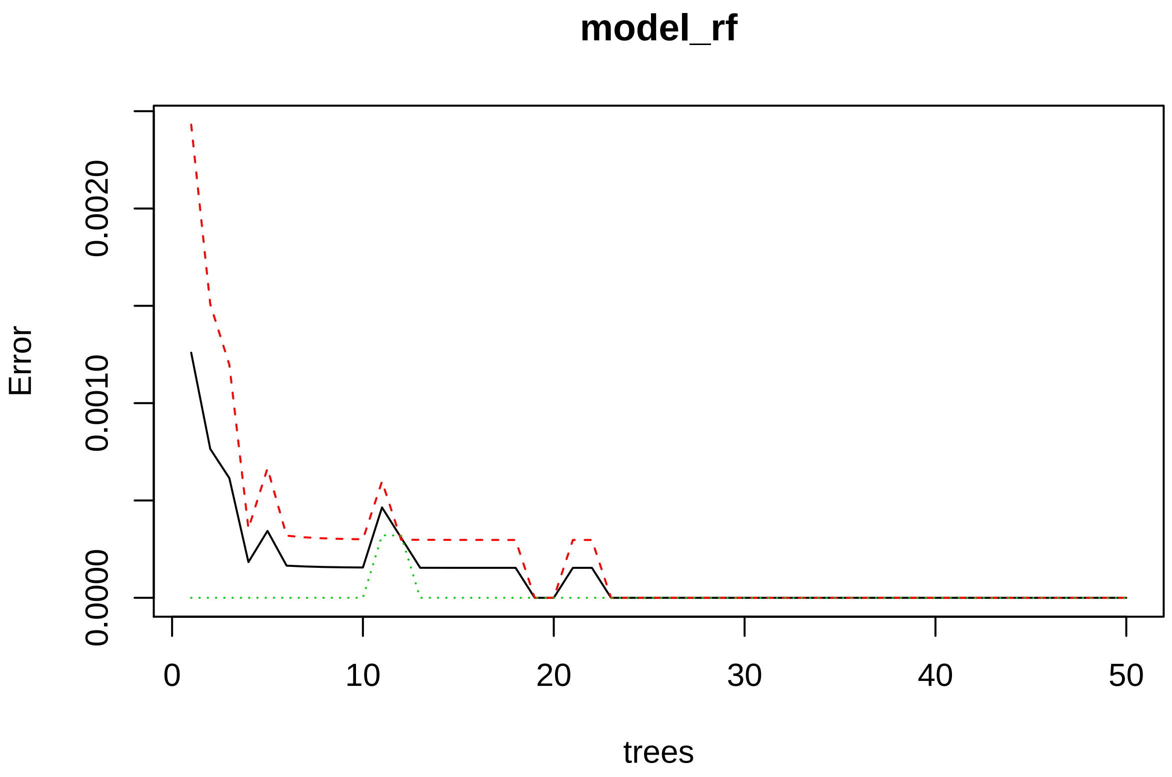
```
## Confusion Matrix and Statistics
##
##            Reference
## Prediction  edible poisonous
##   edible       841         0
##   poisonous      0       783
##
##                Accuracy : 1
##                  95% CI : (0.9977, 1)
##     No Information Rate : 0.5179
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 1
##  Mcnemar's Test P-Value : NA
##
##             Sensitivity : 1.0000
##             Specificity : 1.0000
##          Pos Pred Value : 1.0000
##          Neg Pred Value : 1.0000
##              Prevalence : 0.5179
##          Detection Rate : 0.5179
##    Detection Prevalence : 0.5179
##       Balanced Accuracy : 1.0000
##
##        'Positive' Class : edible
##
```

Perfect prediction here as well.

11.3.3.2 Use of Random Forest

We usually use random forest if a tree is not enough. In this case, as we have perfect prediction using a single tree, it is not really necessary to use a Random Forest algorithm. We just use for learning sake without tuning any of the parameters.

```r
library(randomForest)
model_rf <- randomForest(edibility ~ ., ntree = 50, data = train_mushroom)
plot(model_rf)
```



The default number of trees for the random forest is 500; we just use 50 here. As we can see on the plot, above 20 trees, the error isn't decreasing anymore. And actually, the error seems to be 0 or almost 0.
The next step can tell us this more accurately.

```r
print(model_rf)
```

```
##
## Call:
##  randomForest(formula = edibility ~ ., data = train_mushroom,     ntree = 50)
##                Type of random forest: classification
##                      Number of trees: 50
## No. of variables tried at each split: 4
##
```

```
##         OOB estimate of  error rate: 0%
## Confusion matrix:
##           edible poisonous class.error
## edible      3367         0           0
## poisonous      0      3133           0
```

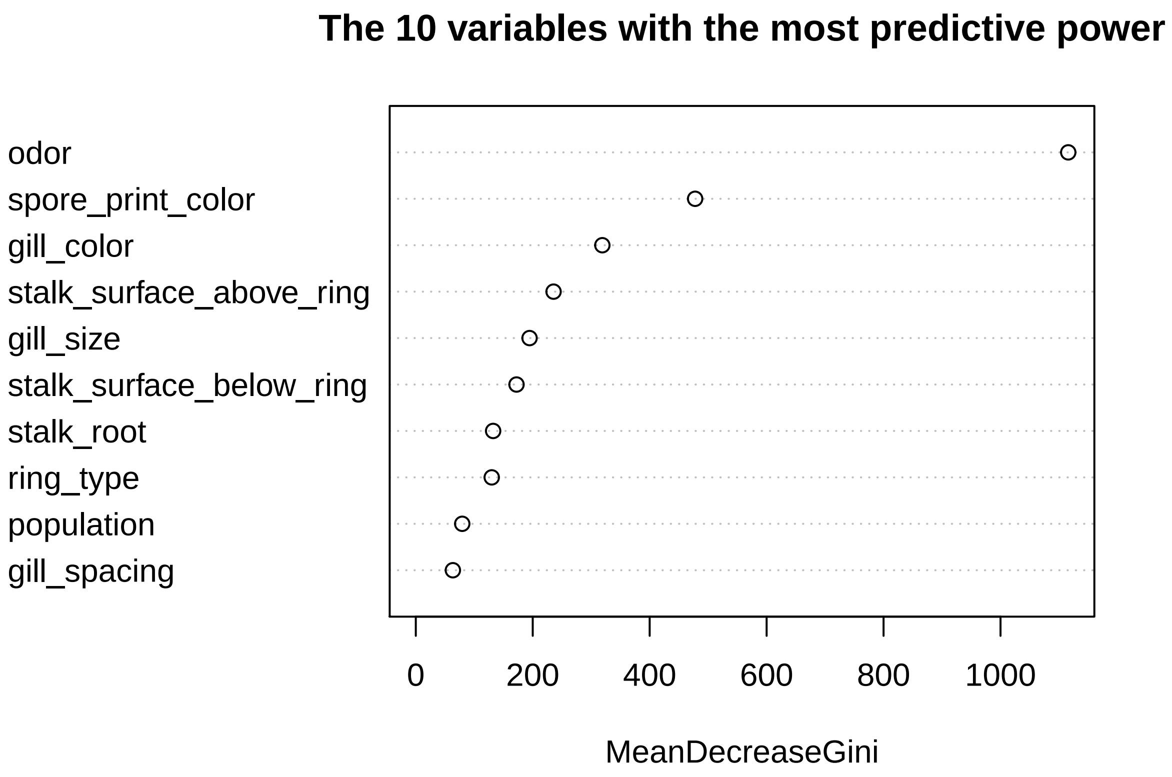
Altough it is not really necessary to this here as we have a perfect prediction, we can use the `confusionMatrix` function from the `caret` pacakge.

```
caret::confusionMatrix(data = model_rf$predicted, reference = train_mushroom$edibility
                       positive = "edible")
```

```
## Confusion Matrix and Statistics
##
##            Reference
## Prediction  edible poisonous
##   edible      3367         0
##   poisonous      0      3133
##
##                Accuracy : 1
##                  95% CI : (0.9994, 1)
##     No Information Rate : 0.518
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 1
##  Mcnemar's Test P-Value : NA
##
##             Sensitivity : 1.000
##             Specificity : 1.000
##          Pos Pred Value : 1.000
##          Neg Pred Value : 1.000
##              Prevalence : 0.518
##          Detection Rate : 0.518
##    Detection Prevalence : 0.518
##       Balanced Accuracy : 1.000
##
##        'Positive' Class : edible
##
```

If we want to look at the most important variable in terms of predicting edibility in our model, we can do that using the *Mean Decreasing Gini*

```
varImpPlot(model_rf, sort = TRUE,
           n.var = 10, main = "The 10 variables with the most predictive power")
```



Another way to look at the predictible power of the variables is to use the `importance` extractor function.

```r
library(tibble)
importance(model_rf) %>% data.frame() %>%
  rownames_to_column(var = "Variable") %>%
  arrange(desc(MeanDecreaseGini)) %>%
  head(10)
```

```
##                    Variable MeanDecreaseGini
## 1                      odor       1115.85522
## 2         spore_print_color        477.71557
## 3                gill_color        319.02467
## 4  stalk_surface_above_ring        235.59574
## 5                 gill_size        194.56155
## 6  stalk_surface_below_ring        172.26749
## 7                stalk_root        132.26045
## 8                 ring_type        129.88445
## 9                population         79.42030
## 10             gill_spacing         63.42436
```

We could compare that with the important variables from the classification tree obtained above.

```r
model_tree_penalty$variable.importance %>%
  as_tibble() %>% rownames_to_column(var = "variable") %>%
  arrange(desc(value)) %>% head(10)
```

```
## # A tibble: 10 x 2
##                    variable     value
##                       <chr>     <dbl>
##  1                     odor 848.00494
##  2        spore_print_color 804.39831
##  3               gill_color 503.71270
##  4 stalk_surface_above_ring 501.28385
##  5 stalk_surface_below_ring 453.92877
##  6                ring_type 450.29286
##  7              ring_number 170.56141
##  8               stalk_root 117.78800
##  9                  habitat  98.22176
## 10   stalk_color_below_ring  74.72602
```

Interestingly gill_size which is the 5th most important predictor in the random forest does not appear in the top 10 of our classification tree.

Now we apply our model to our testing set.

```
test_rf <- predict(model_rf, newdata = test_mushroom)

# Quick check on our prediction
table(test_rf, test_mushroom$edibility)
```

```
##
## test_rf     edible poisonous
##   edible       841         0
##   poisonous      0       783
```

Perfect Prediction!

11.3.3.3 Use of SVM

```
library(e1071)
model_svm <- svm(edibility ~. , data=train_mushroom, cost = 1000, gamma = 0.01)
```

Check the prediction

```
test_svm <- predict(model_svm, newdata = test_mushroom)

table(test_svm, test_mushroom$edibility)
```

```
##
## test_svm    edible poisonous
##   edible       841         0
##   poisonous      0       783
```

And perfect prediction again!

11.4 Communication

With some fine tuning, a regression tree managed to predict accurately the edibility of mushroom. They were 2 parameters to look at: the `cp`and the penalty matrix. Random Forest and SVM achieved similar results out of the box.
The regression tree approach has to be prefered as it is a lot easier to grasp the results from a tree than from a SVM algorithm.

For sure I will take my little tree picture next time I go shrooming. That said, I will still only go with a good mycologist.

Chapter 12

Case Study - Wisconsin Breast Cancer

This is another classification example. We have to classify breast tumors as malign or benign.

The dataset is available on the UCI Machine learning website as well as on [Kaggle](https://www.kaggle.com/uciml/breast-cancer-wisconsin-data.

We have taken ideas from several blogs listed below in the reference section.

12.1 Import the data

```
library(tidyverse)
df <- read_csv("dataset/BreastCancer.csv")

# This is defintely an most important step:
# Check for appropriate class on each of the variable.
glimpse(df)
```

```
## Observations: 569
## Variables: 32
## $ id                      <int> 842302, 842517, 84300903, 84348301, 84...
## $ diagnosis               <chr> "M", "M", "M", "M", "M", "M", "M", "M"...
## $ radius_mean             <dbl> 17.990, 20.570, 19.690, 11.420, 20.290...
## $ texture_mean            <dbl> 10.38, 17.77, 21.25, 20.38, 14.34, 15....
## $ perimeter_mean          <dbl> 122.80, 132.90, 130.00, 77.58, 135.10,...
## $ area_mean               <dbl> 1001.0, 1326.0, 1203.0, 386.1, 1297.0,...
## $ smoothness_mean         <dbl> 0.11840, 0.08474, 0.10960, 0.14250, 0....
## $ compactness_mean        <dbl> 0.27760, 0.07864, 0.15990, 0.28390, 0....
## $ concavity_mean          <dbl> 0.30010, 0.08690, 0.19740, 0.24140, 0....
## $ concave_points_mean     <dbl> 0.14710, 0.07017, 0.12790, 0.10520, 0....
## $ symmetry_mean           <dbl> 0.2419, 0.1812, 0.2069, 0.2597, 0.1809...
## $ fractal_dimension_mean  <dbl> 0.07871, 0.05667, 0.05999, 0.09744, 0....
## $ radius_se               <dbl> 1.0950, 0.5435, 0.7456, 0.4956, 0.7572...
## $ texture_se              <dbl> 0.9053, 0.7339, 0.7869, 1.1560, 0.7813...
## $ perimeter_se            <dbl> 8.589, 3.398, 4.585, 3.445, 5.438, 2.2...
## $ area_se                 <dbl> 153.40, 74.08, 94.03, 27.23, 94.44, 27...
## $ smoothness_se           <dbl> 0.006399, 0.005225, 0.006150, 0.009110...
## $ compactness_se          <dbl> 0.049040, 0.013080, 0.040060, 0.074580...
## $ concavity_se            <dbl> 0.05373, 0.01860, 0.03832, 0.05661, 0....
## $ concave_points_se       <dbl> 0.015870, 0.013400, 0.020580, 0.018670...
## $ symmetry_se             <dbl> 0.03003, 0.01389, 0.02250, 0.05963, 0....
## $ fractal_dimension_se    <dbl> 0.006193, 0.003532, 0.004571, 0.009208...
## $ radius_worst            <dbl> 25.38, 24.99, 23.57, 14.91, 22.54, 15....
## $ texture_worst           <dbl> 17.33, 23.41, 25.53, 26.50, 16.67, 23....
## $ perimeter_worst         <dbl> 184.60, 158.80, 152.50, 98.87, 152.20,...
## $ area_worst              <dbl> 2019.0, 1956.0, 1709.0, 567.7, 1575.0,...
## $ smoothness_worst        <dbl> 0.1622, 0.1238, 0.1444, 0.2098, 0.1374...
## $ compactness_worst       <dbl> 0.6656, 0.1866, 0.4245, 0.8663, 0.2050...
## $ concavity_worst         <dbl> 0.71190, 0.24160, 0.45040, 0.68690, 0....
## $ concave_points_worst    <dbl> 0.26540, 0.18600, 0.24300, 0.25750, 0....
## $ symmetry_worst          <dbl> 0.4601, 0.2750, 0.3613, 0.6638, 0.2364...
## $ fractal_dimension_worst <dbl> 0.11890, 0.08902, 0.08758, 0.17300, 0....
```

So we have 569 observations with 32 variables. Ideally for so many variables, it would be appropriate to get a few more observations.

12.2 Tidy the data

Basics change of variable type for the outcome variable and renaming of variables badly encoded

```
df$diagnosis <- as.factor(df$diagnosis)

#df <- df %>% rename(concave_points_mean = `concave points_mean`,
#                      concave_points_se = `concave points_se`,
#                      concave_points_worst = `concave points_worst`)
```

As you might have noticed, in this case and the precedent we had very little to do here. This is not usually the case.

12.3 Understand the data

This is the circular phase of our dealing with data. This is where each of the transforming, visualizing and modeling stage reinforce each other to create a better understanding.

Check for missing values

```
map_int(df, function(.x) sum(is.na(.x)))
```

```
##                      id               diagnosis             radius_mean 
##                       0                       0                       0 
##            texture_mean          perimeter_mean               area_mean 
##                       0                       0                       0 
##         smoothness_mean        compactness_mean          concavity_mean 
##                       0                       0                       0 
##     concave_points_mean           symmetry_mean  fractal_dimension_mean 
##                       0                       0                       0 
##               radius_se              texture_se            perimeter_se 
##                       0                       0                       0 
##                 area_se           smoothness_se          compactness_se 
##                       0                       0                       0 
##            concavity_se       concave_points_se             symmetry_se 
##                       0                       0                       0 
##    fractal_dimension_se            radius_worst           texture_worst 
##                       0                       0                       0 
##         perimeter_worst              area_worst        smoothness_worst 
##                       0                       0                       0 
##       compactness_worst         concavity_worst    concave_points_worst 
##                       0                       0                       0 
##          symmetry_worst fractal_dimension_worst 
##                       0                       0
```

Good news, there are no missing values.

In the case that there would be many missing values, we would go on the transforming data for some appropriate imputation.

Let's check how balanced is our response variable

```
round(prop.table(table(df$diagnosis)), 2)
```

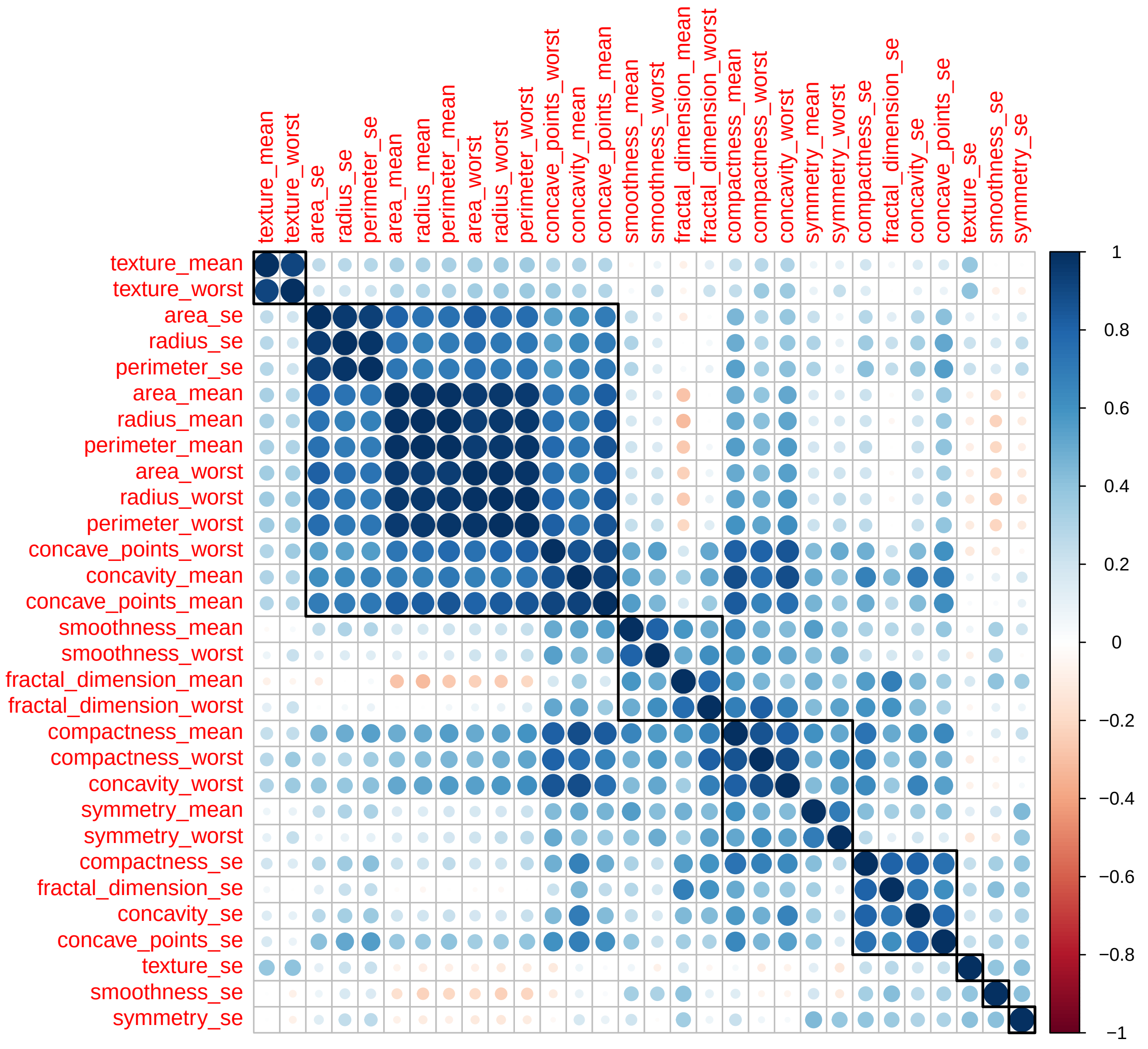
```
## 
##    B    M 
## 0.63 0.37
```

The response variable is slightly unbalanced.

Let's look for correlation in the variables. Most ML algorithms assumed that the predictor variables are independent from each others.

Let's check for correlations. For an anlysis to be robust it is good to remove mutlicollinearity (aka remove highly correlated predictors)

```
df_corr <- cor(df %>% select(-id, -diagnosis))
corrplot::corrplot(df_corr, order = "hclust", tl.cex = 1, addrect = 8)
```



Indeed there are quite a few variables that are correlated. On the next step, we will remove the highly correlated ones using the `caret` package.

12.3.1 Transform the data

```
library(caret)
# The findcorrelation() function from caret package remove highly correlated predictor
# based on whose correlation is above 0.9. This function uses a heuristic algorithm
# to determine which variable should be removed instead of selecting blindly
df2 <- df %>% select(-findCorrelation(df_corr, cutoff = 0.9))

#Number of columns for our new data frame
ncol(df2)
```

```
## [1] 22
```

So our new data frame `df2` is 10 variables shorter.

12.3.2 Pre-process the data

12.3.2.1 Using PCA

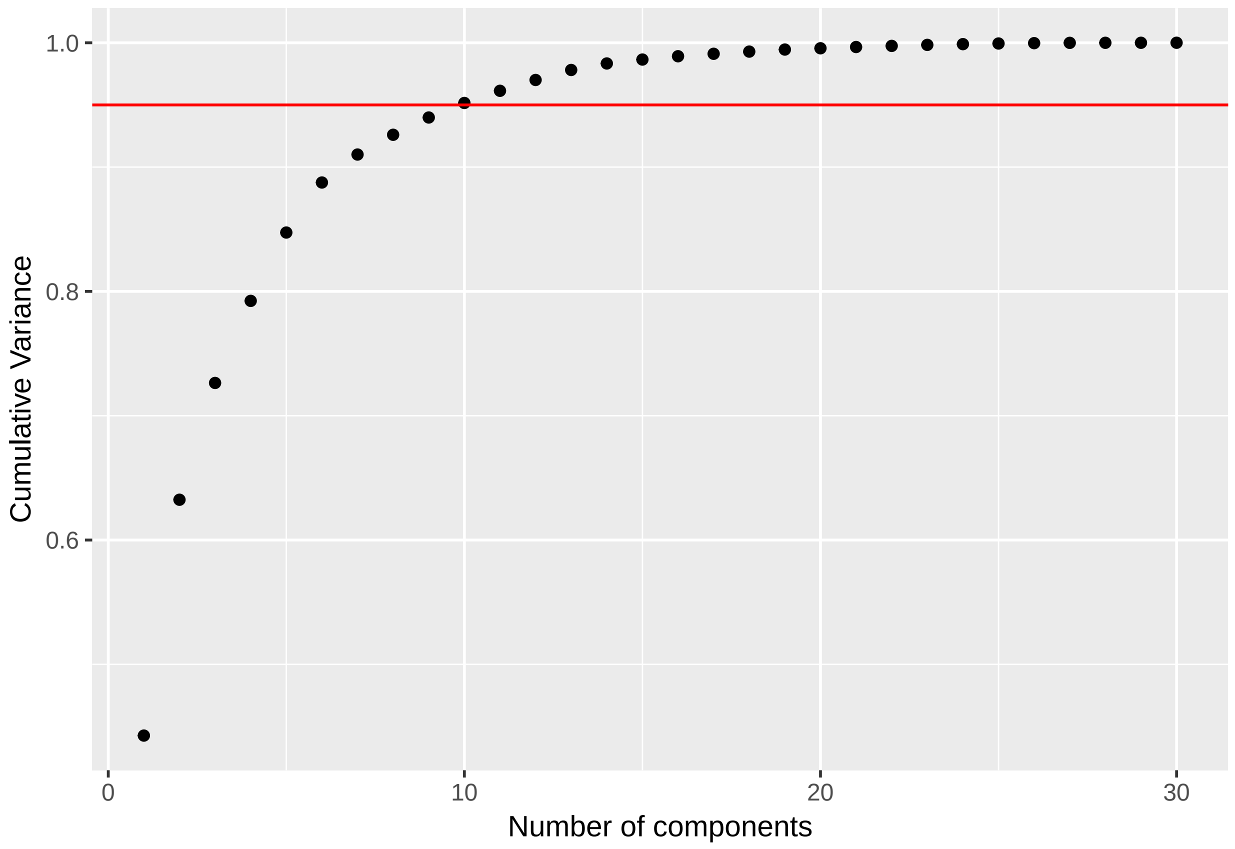
Let's first go on an unsupervised analysis with a PCA analysis.
To do so, we will remove the `id` and `diagnosis` variable, then we will also scale and ceter the variables.

```
preproc_pca_df <- prcomp(df %>% select(-id, -diagnosis), scale = TRUE, center = TRUE)
summary(preproc_pca_df)
```

```
## Importance of components:
##                           PC1    PC2     PC3     PC4     PC5     PC6
## Standard deviation     3.6444 2.3857 1.67867 1.40735 1.28403 1.09880
## Proportion of Variance 0.4427 0.1897 0.09393 0.06602 0.05496 0.04025
## Cumulative Proportion  0.4427 0.6324 0.72636 0.79239 0.84734 0.88759
##                            PC7     PC8    PC9    PC10   PC11    PC12
## Standard deviation     0.82172 0.69037 0.6457 0.59219 0.5421 0.51104
## Proportion of Variance 0.02251 0.01589 0.0139 0.01169 0.0098 0.00871
## Cumulative Proportion  0.91010 0.92598 0.9399 0.95157 0.9614 0.97007
##                           PC13    PC14    PC15    PC16    PC17    PC18
## Standard deviation     0.49128 0.39624 0.30681 0.28260 0.24372 0.22939
## Proportion of Variance 0.00805 0.00523 0.00314 0.00266 0.00198 0.00175
## Cumulative Proportion  0.97812 0.98335 0.98649 0.98915 0.99113 0.99288
##                           PC19    PC20   PC21    PC22    PC23   PC24
## Standard deviation     0.22244 0.17652 0.1731 0.16565 0.15602 0.1344
## Proportion of Variance 0.00165 0.00104 0.0010 0.00091 0.00081 0.0006
## Cumulative Proportion  0.99453 0.99557 0.9966 0.99749 0.99830 0.9989
##                           PC25    PC26    PC27    PC28    PC29    PC30
## Standard deviation     0.12442 0.09043 0.08307 0.03987 0.02736 0.01153
## Proportion of Variance 0.00052 0.00027 0.00023 0.00005 0.00002 0.00000
## Cumulative Proportion  0.99942 0.99969 0.99992 0.99997 1.00000 1.00000
```

```
# Calculate the proportion of variance explained
pca_df_var <- preproc_pca_df$sdev^2
```

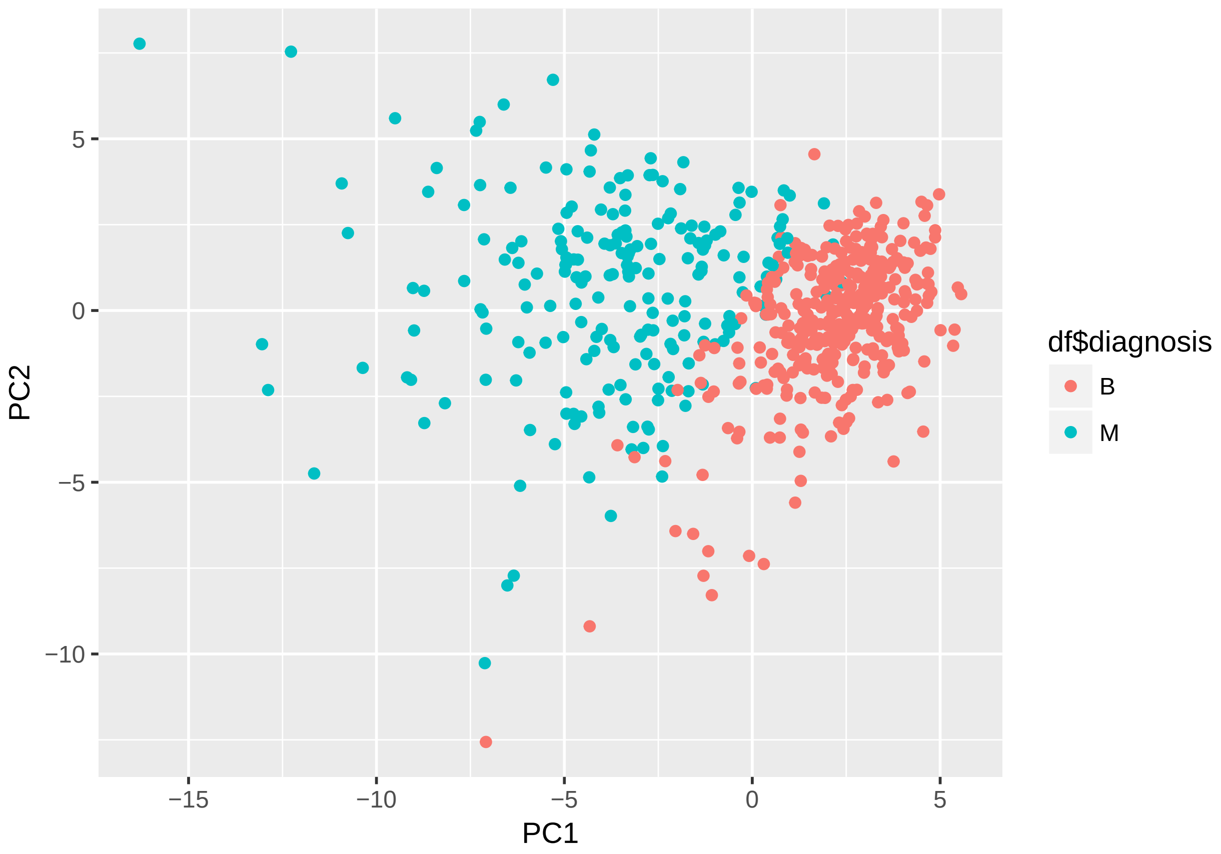
```r
pve_df <- pca_df_var / sum(pca_df_var)
cum_pve <- cumsum(pve_df)
pve_table <- tibble(comp = seq(1:ncol(df %>% select(-id, -diagnosis))), pve_df, cum_pve

ggplot(pve_table, aes(x = comp, y = cum_pve)) +
  geom_point() +
  geom_abline(intercept = 0.95, color = "red", slope = 0) +
  labs(x = "Number of components", y = "Cumulative Variance")
```



With the original dataset, 95% of the variance is explained with 10 PC's.

Let's check on the most influential variables for the first 2 components

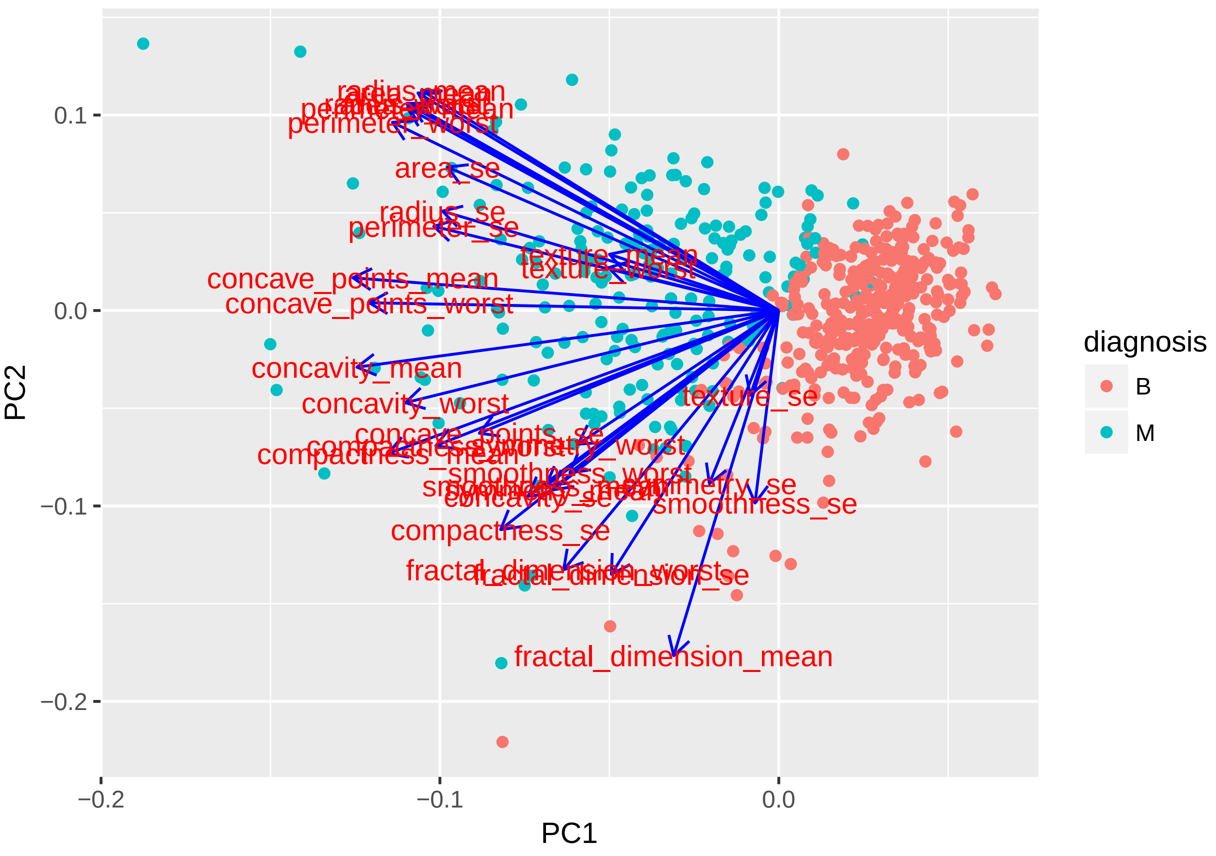
```r
pca_df <- as_tibble(preproc_pca_df$x)

ggplot(pca_df, aes(x = PC1, y = PC2, col = df$diagnosis)) + geom_point()
```



It does look like the first 2 components managed to separate the diagnosis quite well. Lots of potential here.

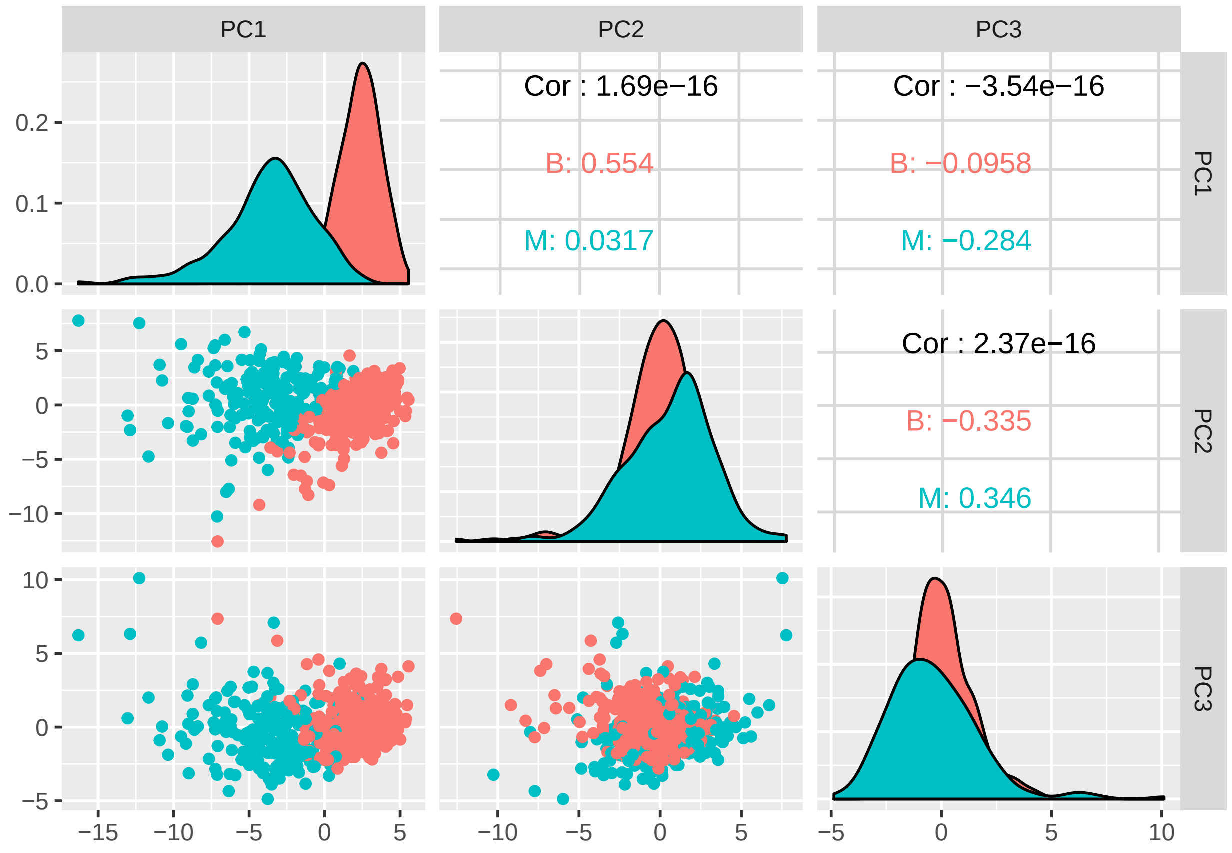
If we want to get a more detailled analysis of what variables are the most influential in the first 2 components, we can use the `ggfortify` library.

```r
library(ggfortify)
autoplot(preproc_pca_df, data = df,  colour = 'diagnosis',
                    loadings = FALSE, loadings.label = TRUE, loadings.colour = "blue")
```



Let's visuzalize the first 3 components.

```
df_pcs <- cbind(as_tibble(df$diagnosis), as_tibble(preproc_pca_df$x))
GGally::ggpairs(df_pcs, columns = 2:4, ggplot2::aes(color = value))
```

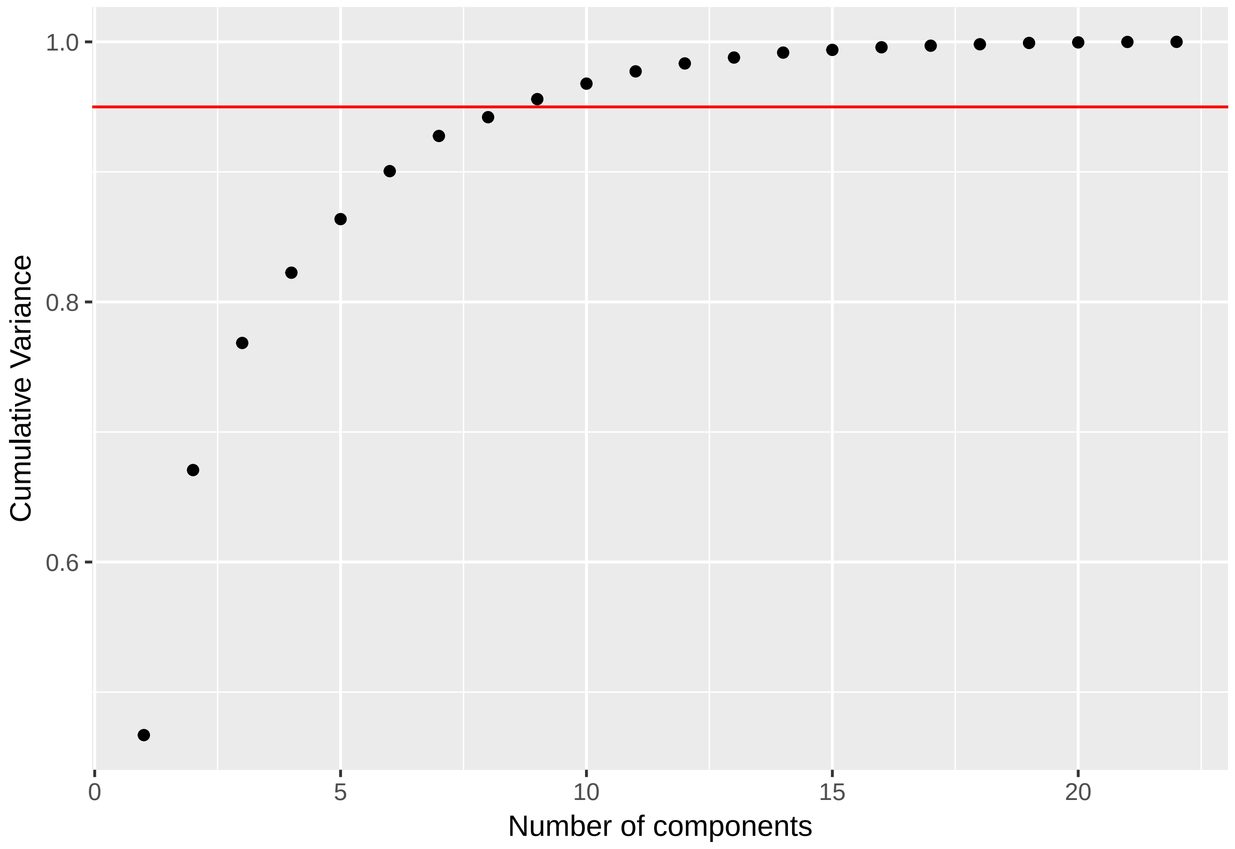


Let's do the same exercise with our second df, the one where we removed the highly correlated predictors.

```
preproc_pca_df2 <- prcomp(df2, scale = TRUE, center = TRUE)
summary(preproc_pca_df2)
```

```
## Importance of components:
##                           PC1    PC2     PC3     PC4     PC5     PC6
## Standard deviation     3.2051 2.1175 1.46634 1.09037 0.95215 0.90087
## Proportion of Variance 0.4669 0.2038 0.09773 0.05404 0.04121 0.03689
## Cumulative Proportion  0.4669 0.6707 0.76847 0.82251 0.86372 0.90061
##                            PC7     PC8    PC9    PC10    PC11    PC12
## Standard deviation     0.77121 0.56374 0.5530 0.51130 0.45605 0.36602
## Proportion of Variance 0.02703 0.01445 0.0139 0.01188 0.00945 0.00609
## Cumulative Proportion  0.92764 0.94209 0.9560 0.96787 0.97732 0.98341
##                           PC13    PC14   PC15   PC16    PC17   PC18   PC19
## Standard deviation     0.31602 0.28856 0.2152 0.2098 0.16346 0.1558 0.1486
## Proportion of Variance 0.00454 0.00378 0.0021 0.0020 0.00121 0.0011 0.0010
## Cumulative Proportion  0.98795 0.99174 0.9938 0.9958 0.99706 0.9982 0.9992
##                           PC20    PC21    PC22
## Standard deviation     0.09768 0.08667 0.03692
## Proportion of Variance 0.00043 0.00034 0.00006
## Cumulative Proportion  0.99960 0.99994 1.00000
```

```r
pca_df2_var <- preproc_pca_df2$sdev^2

# proportion of variance explained
pve_df2 <- pca_df2_var / sum(pca_df2_var)
cum_pve_df2 <- cumsum(pve_df2)
pve_table_df2 <- tibble(comp = seq(1:ncol(df2)), pve_df2, cum_pve_df2)

ggplot(pve_table_df2, aes(x = comp, y = cum_pve_df2)) +
  geom_point() +
  geom_abline(intercept = 0.95, color = "red", slope = 0) +
  labs(x = "Number of components", y = "Cumulative Variance")
```



In this case, around 8 PC's explained 95% of the variance.

12.3.2.2 Using LDA

The advantage of using LDA is that it takes into consideration the different class.

```r
preproc_lda_df <- MASS::lda(diagnosis ~., data = df, center = TRUE, scale = TRUE)
preproc_lda_df
```

```
## Call:
```

```
## lda(diagnosis ~ ., data = df, center = TRUE, scale = TRUE)
##
## Prior probabilities of groups:
##         B         M
## 0.6274165 0.3725835
##
## Group means:
##         id radius_mean texture_mean perimeter_mean area_mean
## B 26543825    12.14652     17.91476       78.07541  462.7902
## M 36818050    17.46283     21.60491      115.36538  978.3764
##   smoothness_mean compactness_mean concavity_mean concave_points_mean
## B      0.09247765       0.08008462     0.04605762          0.02571741
## M      0.10289849       0.14518778     0.16077472          0.08799000
##   symmetry_mean fractal_dimension_mean radius_se texture_se perimeter_se
## B      0.174186             0.06286739 0.2840824   1.220380     2.000321
## M      0.192909             0.06268009 0.6090825   1.210915     4.323929
##    area_se smoothness_se compactness_se concavity_se concave_points_se
## B 21.13515   0.007195902     0.02143825   0.02599674       0.009857653
## M 72.67241   0.006780094     0.03228117   0.04182401       0.015060472
##   symmetry_se fractal_dimension_se radius_worst texture_worst
## B  0.02058381          0.003636051     13.37980      23.51507
## M  0.02047240          0.004062406     21.13481      29.31821
##   perimeter_worst area_worst smoothness_worst compactness_worst
## B        87.00594   558.8994        0.1249595         0.1826725
## M       141.37033  1422.2863        0.1448452         0.3748241
##   concavity_worst concave_points_worst symmetry_worst
## B       0.1662377           0.07444434      0.2702459
## M       0.4506056           0.18223731      0.3234679
##   fractal_dimension_worst
## B              0.07944207
## M              0.09152995
##
## Coefficients of linear discriminants:
##                                  LD1
## id                      -2.512117e-10
## radius_mean             -1.080876e+00
## texture_mean             2.338408e-02
## perimeter_mean           1.172707e-01
## area_mean                1.595690e-03
## smoothness_mean          5.251575e-01
## compactness_mean        -2.094197e+01
## concavity_mean           6.955923e+00
## concave_points_mean      1.047567e+01
## symmetry_mean            4.938898e-01
## fractal_dimension_mean  -5.937663e-02
## radius_se                2.101503e+00
## texture_se              -3.979869e-02
## perimeter_se            -1.121814e-01
## area_se                 -4.083504e-03
## smoothness_se            7.987663e+01
## compactness_se           1.387026e-01
## concavity_se            -1.768261e+01
## concave_points_se        5.350520e+01
## symmetry_se              8.143611e+00
## fractal_dimension_se    -3.431356e+01
## radius_worst             9.677207e-01
## texture_worst            3.540591e-02
## perimeter_worst         -1.204507e-02
## area_worst              -5.012127e-03
## smoothness_worst         2.612258e+00
## compactness_worst        3.636892e-01
## concavity_worst          1.880699e+00
## concave_points_worst     2.218189e+00
## symmetry_worst           2.783102e+00
## fractal_dimension_worst  2.117830e+01
```

```
# Making a df out of the LDA for visualization purpose.
predict_lda_df <- predict(preproc_lda_df, df)$x %>%
  as_data_frame() %>%
  cbind(diagnosis = df$diagnosis)

glimpse(predict_lda_df)
```

```
## Observations: 569
## Variables: 2
## $ LD1       <dbl> 3.3257395, 2.3298023, 3.7416859, 4.0209903, 2.275428...
## $ diagnosis <fctr> M, M, M, M, M, M, M, M, M, M, M, M, M, M, M, M, M, ...
```

12.3.3 Model the data

Let's first create a testing and training set using `caret`

```
set.seed(1815)
df3 <- cbind(diagnosis = df$diagnosis, df2)
df_sampling_index <- createDataPartition(df3$diagnosis, times = 1, p = 0.8, list = FALSE
df_training <- df3[df_sampling_index, ]
df_testing <-  df3[-df_sampling_index, ]
df_control <- trainControl(method="cv",
                           number = 15,
                           classProbs = TRUE,
                              summaryFunction = twoClassSummary)
```

12.3.3.1 Logistic regression

Our first model is doing logistic regression on `df2`, the data frame where we took away the highly correlated variables.

```
model_logreg_df <- train(diagnosis ~., data = df_training, method = "glm",
                         metric = "ROC", preProcess = c("scale", "center"),
                         trControl = df_control)

prediction_logreg_df <- predict(model_logreg_df, df_testing)
cm_logreg_df <- confusionMatrix(prediction_logreg_df, df_testing$diagnosis, positive = '
cm_logreg_df
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  B  M
##          B 71  2
##          M  0 40
##
##                Accuracy : 0.9823
##                  95% CI : (0.9375, 0.9978)
##     No Information Rate : 0.6283
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.9617
##  Mcnemar's Test P-Value : 0.4795
##
##             Sensitivity : 0.9524
##             Specificity : 1.0000
##          Pos Pred Value : 1.0000
##          Neg Pred Value : 0.9726
##              Prevalence : 0.3717
##          Detection Rate : 0.3540
##    Detection Prevalence : 0.3540
##       Balanced Accuracy : 0.9762
##
##        'Positive' Class : M
##
```

12.3.3.2 Random Forest

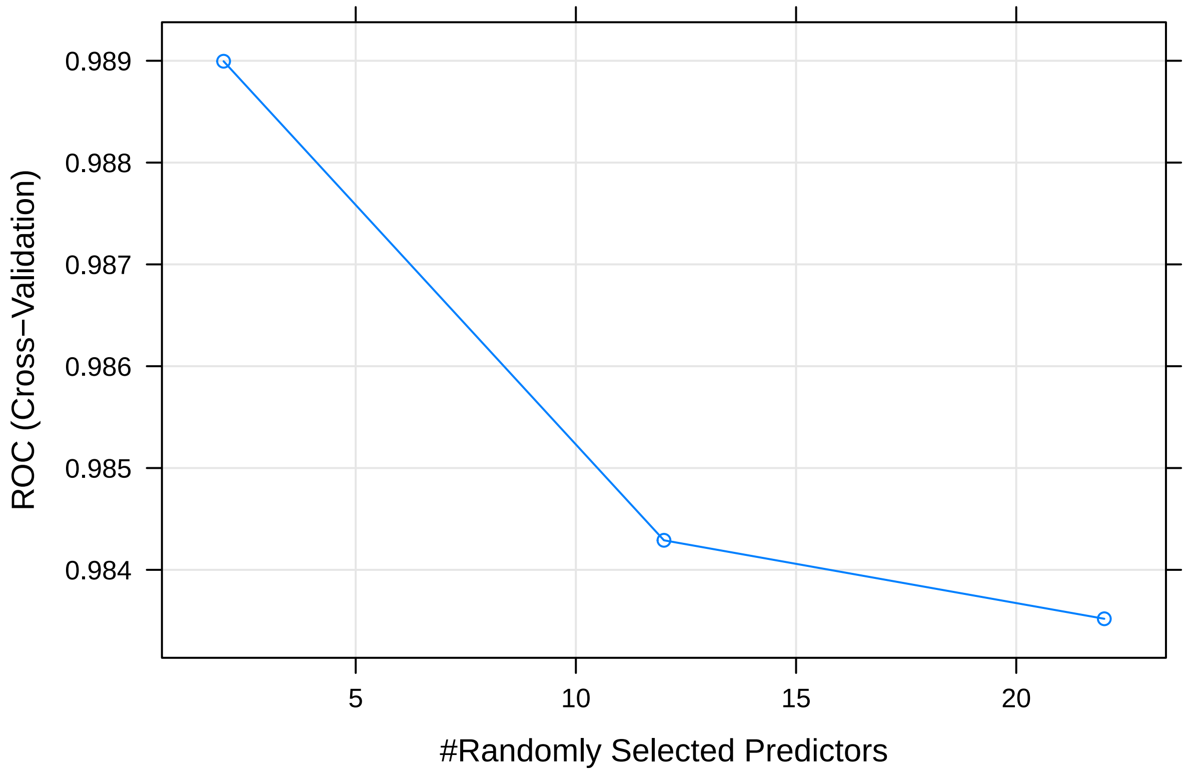
Our second model uses random forest. Similarly, we using the `df2` data frame, the one where we took away the highly correlated variables.

```
model_rf_df <- train(diagnosis ~., data = df_training,
                     method = "rf",
                     metric = 'ROC',
                     trControl = df_control)

prediction_rf_df <- predict(model_rf_df, df_testing)
cm_rf_df <- confusionMatrix(prediction_rf_df, df_testing$diagnosis, positive = "M")
cm_rf_df
```

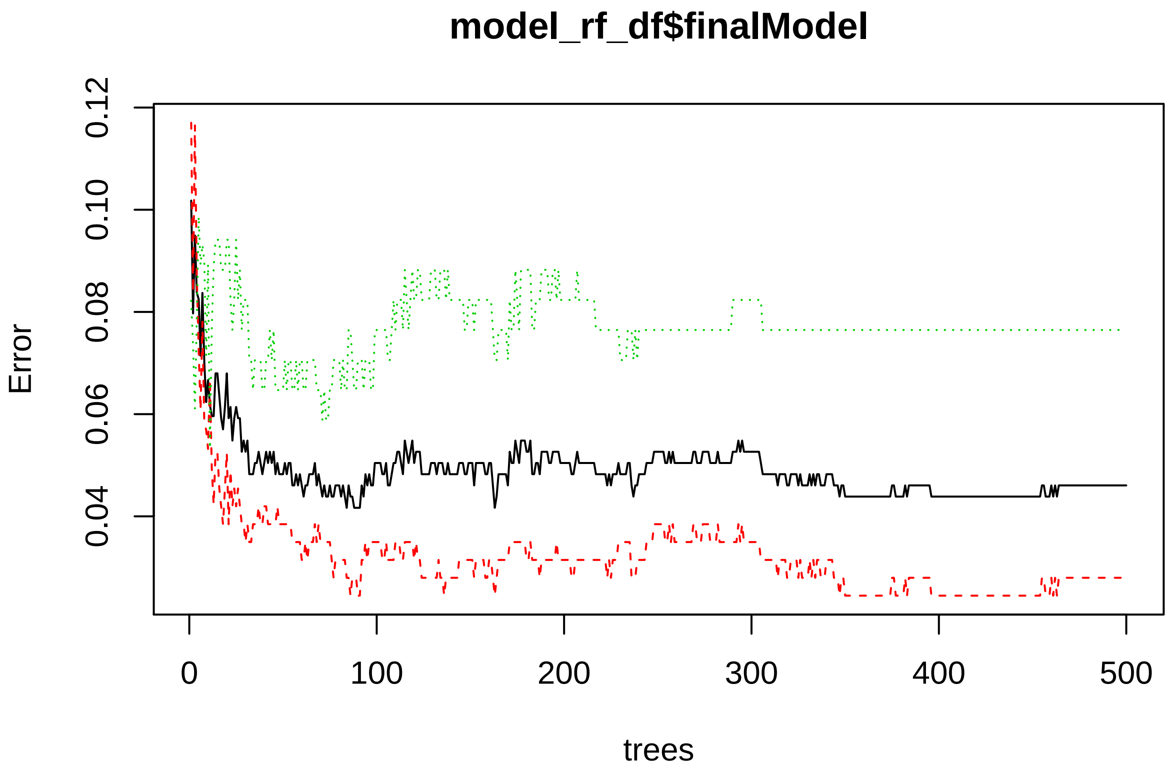
```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  B  M
##          B 71  3
##          M  0 39
##
##                Accuracy : 0.9735
##                  95% CI : (0.9244, 0.9945)
##     No Information Rate : 0.6283
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.9423
##  Mcnemar's Test P-Value : 0.2482
##
##             Sensitivity : 0.9286
##             Specificity : 1.0000
##          Pos Pred Value : 1.0000
##          Neg Pred Value : 0.9595
##              Prevalence : 0.3717
##          Detection Rate : 0.3451
##    Detection Prevalence : 0.3451
##       Balanced Accuracy : 0.9643
##
##        'Positive' Class : M
##
```

Let's make some diagnostic plots.

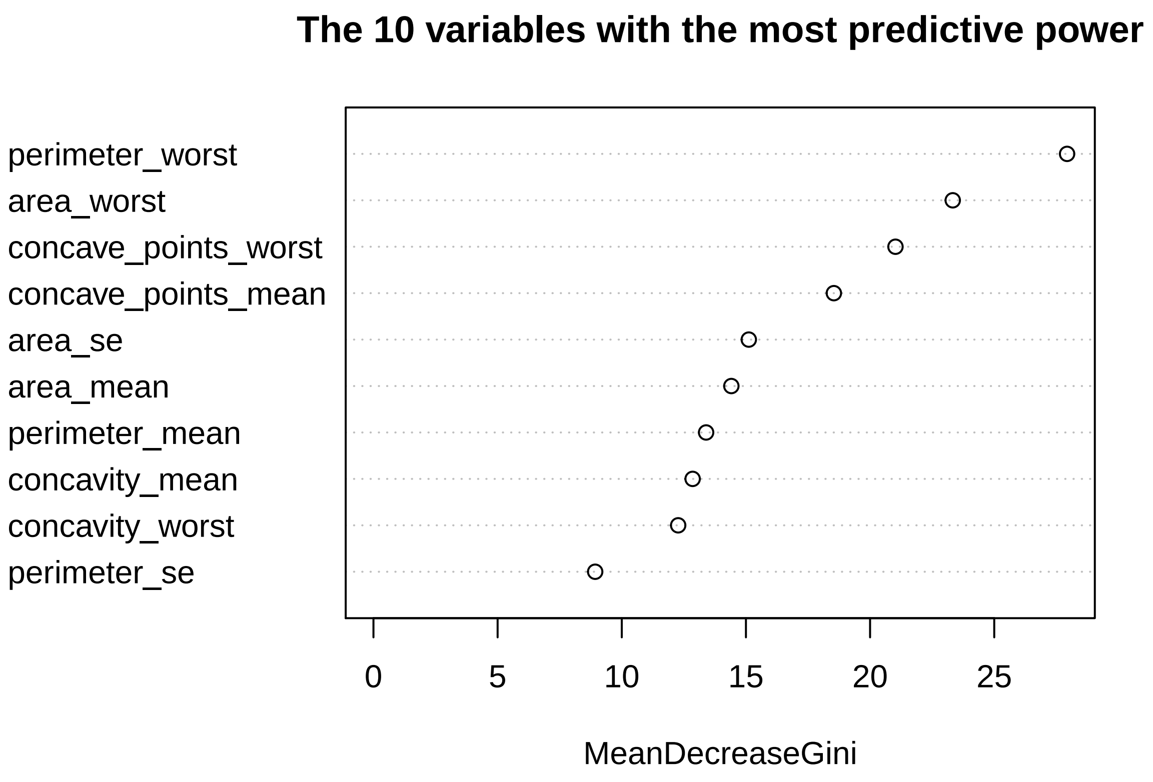
```r
plot(model_rf_df)
```



```r
plot(model_rf_df$finalModel)
```

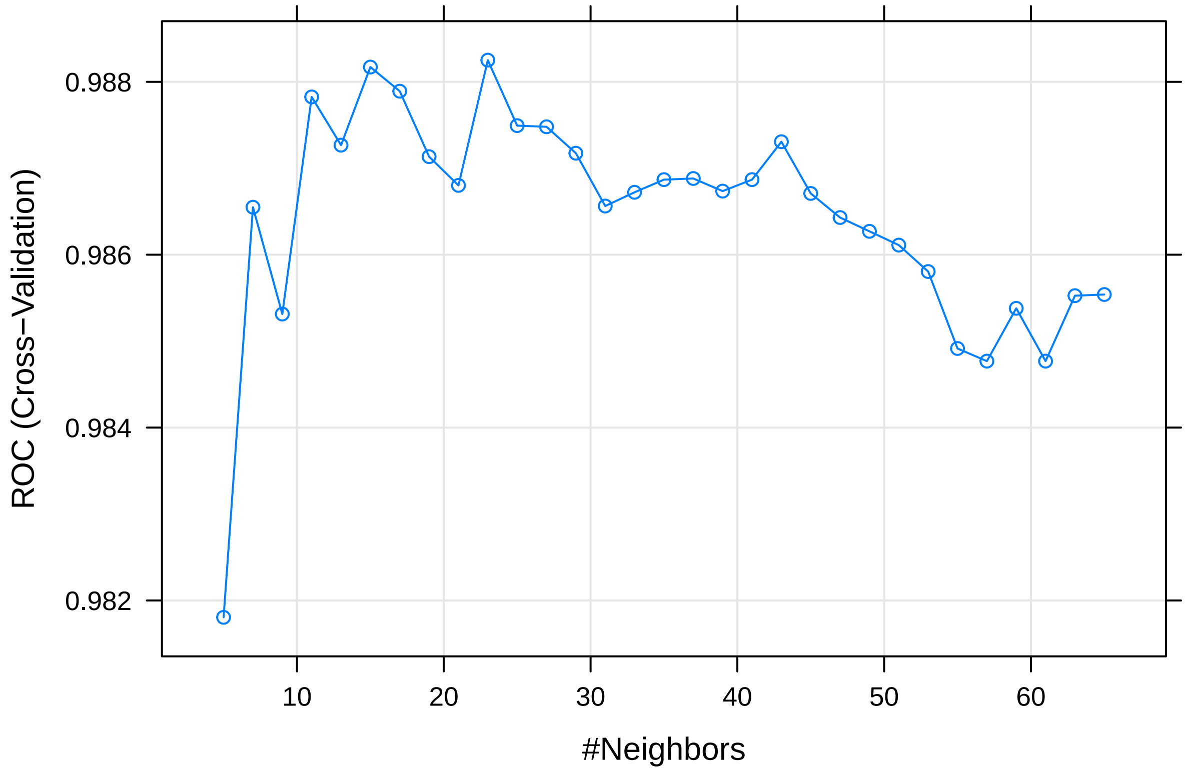


```r
varImpPlot(model_rf_df$finalModel, sort = TRUE,
           n.var = 10, main = "The 10 variables with the most predictive power")
```



12.3.3.3 KNN

```r
model_knn_df <- train(diagnosis ~., data = df_training,
                      method = "knn",
                      metric = "ROC",
                      preProcess = c("scale", "center"),
                      trControl = df_control,
                      tuneLength =31)

plot(model_knn_df)
```



```r
prediction_knn_df <- predict(model_knn_df, df_testing)
cm_knn_df <- confusionMatrix(prediction_knn_df, df_testing$diagnosis, positive = "M")
cm_knn_df
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  B  M
##          B 70  6
##          M  1 36
##
##                Accuracy : 0.9381
##                  95% CI : (0.8765, 0.9747)
##     No Information Rate : 0.6283
##     P-Value [Acc > NIR] : 1.718e-14
##
##                   Kappa : 0.8641
##  Mcnemar's Test P-Value : 0.1306
##
##             Sensitivity : 0.8571
##             Specificity : 0.9859
##          Pos Pred Value : 0.9730
##          Neg Pred Value : 0.9211
##              Prevalence : 0.3717
##          Detection Rate : 0.3186
##    Detection Prevalence : 0.3274
##       Balanced Accuracy : 0.9215
##
##        'Positive' Class : M
##
```

12.3.3.4 Support Vector Machine

```
set.seed(1815)
model_svm_df <- train(diagnosis ~., data = df_training, method = "svmLinear",
                      metric = "ROC",
                      preProcess = c("scale", "center"),
                      trace = FALSE,
                      trControl = df_control)

prediction_svm_df <- predict(model_svm_df, df_testing)
cm_svm_df <- confusionMatrix(prediction_svm_df, df_testing$diagnosis, positive = "M")
cm_svm_df
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  B  M
##          B 71  2
##          M  0 40
##
##                Accuracy : 0.9823
##                  95% CI : (0.9375, 0.9978)
##     No Information Rate : 0.6283
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.9617
##  Mcnemar's Test P-Value : 0.4795
##
##             Sensitivity : 0.9524
##             Specificity : 1.0000
##          Pos Pred Value : 1.0000
##          Neg Pred Value : 0.9726
##              Prevalence : 0.3717
##          Detection Rate : 0.3540
##    Detection Prevalence : 0.3540
##       Balanced Accuracy : 0.9762
##
```

```
##        'Positive' Class : M
##
```

This is is an OK model.
I am wondering though if we could achieve better results with SVM when doing it on the PCA data set.

```
set.seed(1815)
df_control_pca <- trainControl(method="cv",
                              number = 15,
                              preProcOptions = list(thresh = 0.9), # threshold for pca
                              classProbs = TRUE,
                              summaryFunction = twoClassSummary)

model_svm_pca_df <- train(diagnosis~.,
                          df_training, method = "svmLinear", metric = "ROC",
                          preProcess = c('center', 'scale', "pca"),
                          trControl = df_control_pca)

prediction_svm_pca_df <- predict(model_svm_pca_df, df_testing)
cm_svm_pca_df <- confusionMatrix(prediction_svm_pca_df, df_testing$diagnosis, positive =
cm_svm_pca_df
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  B  M
##          B 70  2
##          M  1 40
##
##                Accuracy : 0.9735
##                  95% CI : (0.9244, 0.9945)
##     No Information Rate : 0.6283
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.9429
##  Mcnemar's Test P-Value : 1
##
##             Sensitivity : 0.9524
##             Specificity : 0.9859
##          Pos Pred Value : 0.9756
##          Neg Pred Value : 0.9722
##              Prevalence : 0.3717
##          Detection Rate : 0.3540
##    Detection Prevalence : 0.3628
##       Balanced Accuracy : 0.9691
```

```
##
##        'Positive' Class : M
##
```

That's already better. The treshold parameter is what we needed to play with.

12.3.3.5 Neural Network with LDA

To use the LDA pre-processing step, we need to also create the same training and testing set.

```
lda_training <- predict_lda_df[df_sampling_index, ]
lda_testing <- predict_lda_df[-df_sampling_index, ]
model_nnetlda_df <- train(diagnosis ~., lda_training,
                          method = "nnet",
                          metric = "ROC",
                          preProcess = c("center", "scale"),
                          tuneLength = 10,
                          trace = FALSE,
                          trControl = df_control)

prediction_nnetlda_df <- predict(model_nnetlda_df, lda_testing)
cm_nnetlda_df <- confusionMatrix(prediction_nnetlda_df, lda_testing$diagnosis, positive
cm_nnetlda_df
```

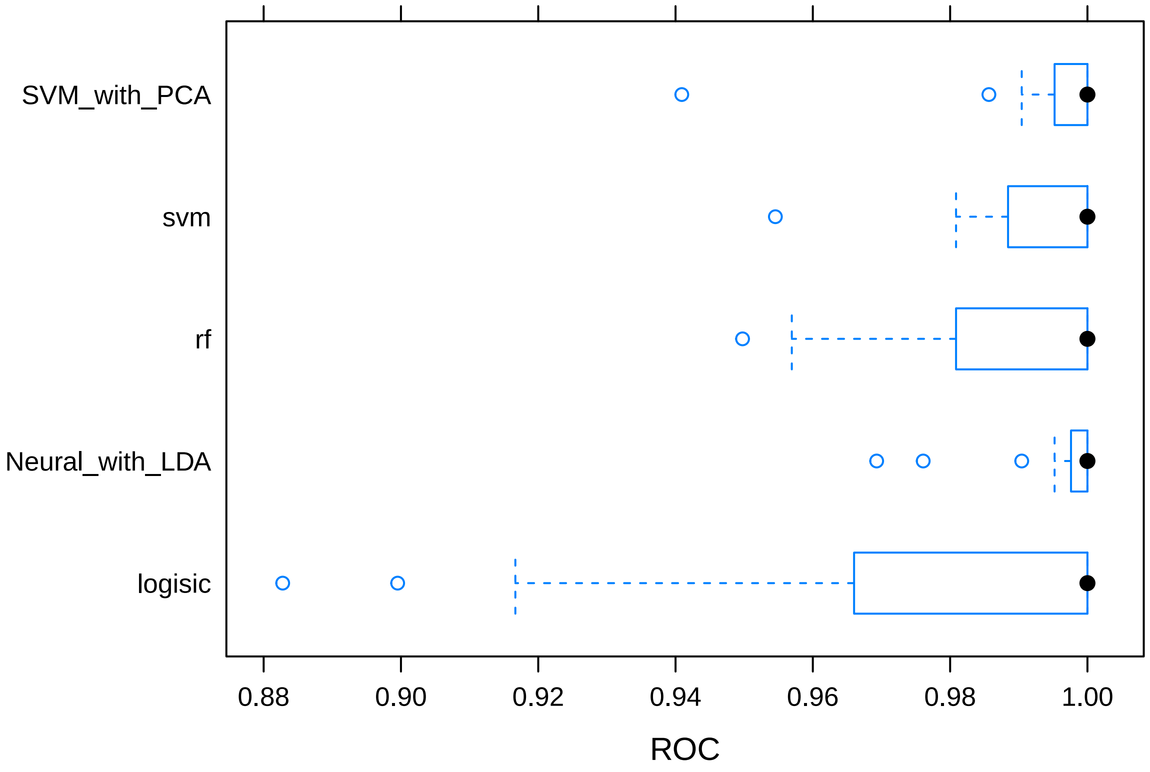
```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  B  M
##          B 71  1
##          M  0 41
##
##                Accuracy : 0.9912
##                  95% CI : (0.9517, 0.9998)
##     No Information Rate : 0.6283
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.981
##  Mcnemar's Test P-Value : 1
##
##             Sensitivity : 0.9762
##             Specificity : 1.0000
##          Pos Pred Value : 1.0000
##          Neg Pred Value : 0.9861
##              Prevalence : 0.3717
##          Detection Rate : 0.3628
##    Detection Prevalence : 0.3628
##       Balanced Accuracy : 0.9881
##
##        'Positive' Class : M
##
```

12.3.3.6 Models evaluation

```
model_list <- list(logisic = model_logreg_df, rf = model_rf_df,
                   svm = model_svm_df, SVM_with_PCA = model_svm_pca_df,
                   Neural_with_LDA = model_nnetlda_df)
results <- resamples(model_list)

summary(results)
```

```
##
## Call:
## summary.resamples(object = results)
##
## Models: logisic, rf, svm, SVM_with_PCA, Neural_with_LDA
## Number of resamples: 15
##
## ROC
##                      Min.   1st Qu. Median      Mean 3rd Qu. Max. NA's
## logisic         0.8827751 0.9660088      1 0.9744418       1    1    0
## rf              0.9497608 0.9808612      1 0.9889952       1    1    0
## svm             0.9545455 0.9884370      1 0.9928761       1    1    0
## SVM_with_PCA    0.9409091 0.9952153      1 0.9932430       1    1    0
## Neural_with_LDA 0.9692982 0.9976077      1 0.9954014       1    1    0
##
## Sens
##                      Min.   1st Qu.    Median      Mean 3rd Qu. Max. NA's
## logisic         0.8947368 0.9473684 0.9473684 0.9615789       1    1    0
## rf              0.8947368 0.9473684 1.0000000 0.9721053       1    1    0
## svm             0.9473684 1.0000000 1.0000000 0.9929825       1    1    0
## SVM_with_PCA    0.9473684 1.0000000 1.0000000 0.9894737       1    1    0
## Neural_with_LDA 0.8947368 1.0000000 1.0000000 0.9859649       1    1    0
##
## Spec
##                      Min.   1st Qu.    Median      Mean 3rd Qu. Max. NA's
## logisic         0.8181818 0.9128788 1.0000000 0.9530303       1    1    0
## rf              0.6363636 0.9090909 0.9090909 0.9095960       1    1    0
## svm             0.8181818 0.9090909 0.9166667 0.9343434       1    1    0
## SVM_with_PCA    0.8181818 0.9090909 1.0000000 0.9580808       1     1     0
## Neural_with_LDA 0.8181818 0.9128788 1.0000000 0.9525253       1     1     0
```

```r
bwplot(results, metric = "ROC")
```



```r
#dotplot(results)
```

The logistic has to much variability for it to be reliable. The Random Forest and Neural Network with LDA pre-processing are giving the best results. The ROC metric measure the auc of the roc curve of each model. This metric is independent of any threshold. Let's remember how these models result with the testing dataset. Prediction classes are obtained by default with a threshold of 0.5 which could not be the best with an unbalanced dataset like this.

```r
cm_list <- list(cm_rf = cm_rf_df, cm_svm = cm_svm_df,
                   cm_logisic = cm_logreg_df, cm_nnet_LDA = cm_nnetlda_df)
results <- map_df(cm_list, function(x) x$byClass) %>% as_tibble() %>%
  mutate(stat = names(cm_rf_df$byClass))

results
```

```
## # A tibble: 11 x 5
##        cm_rf    cm_svm cm_logisic cm_nnet_LDA                 stat
##        <dbl>     <dbl>      <dbl>       <dbl>                <chr>
##  1 0.9285714 0.9523810  0.9523810   0.9761905          Sensitivity
##  2 1.0000000 1.0000000  1.0000000   1.0000000          Specificity
##  3 1.0000000 1.0000000  1.0000000   1.0000000       Pos Pred Value
##  4 0.9594595 0.9726027  0.9726027   0.9861111       Neg Pred Value
##  5 1.0000000 1.0000000  1.0000000   1.0000000            Precision
##  6 0.9285714 0.9523810  0.9523810   0.9761905               Recall
##  7 0.9629630 0.9756098  0.9756098   0.9879518                   F1
##  8 0.3716814 0.3716814  0.3716814   0.3716814           Prevalence
##  9 0.3451327 0.3539823  0.3539823   0.3628319       Detection Rate
## 10 0.3451327 0.3539823  0.3539823   0.3628319 Detection Prevalence
## 11 0.9642857 0.9761905  0.9761905   0.9880952    Balanced Accuracy
```

The best results for sensitivity (detection of breast cases) is LDA_NNET which also has a great F1 score.

12.4 References

A useful popular kernel on this dataset on Kaggle Another one, also on Kaggle And another one, especially nice to compare models.

Index

Bibliography