Machine Learning with R

François de Ryckel

2017-05-12

Contents

Chapter 1

Prerequisites

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 great things I'm learning in a central location.

So this book is a compilation of all the great 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 responsability 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

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 a R book.

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

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'am happy to learn inside out one opiniated framework in order to be able to aplly 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 indepedent variables.

Chapter 2

Tests and inferences

Definetly 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 the few important topics and r-ways to do them.

Let's get started.

You can label chapter and section titles using `{#label}` after them, e.g., we can reference Chapter 3. If you do not manually label them, there will be automatic labels anyway, e.g., Chapter **??**.

Figures and tables with captions will be placed in `figure` and `table` environments, respectively.

```r
par(mar = c(4, 4, .1, .1))
plot(pressure, type = 'b', pch = 19)
```

Reference a figure by its code chunk label with the `fig:` prefix, e.g., see Figure 2.1. Similarly, you can reference tables generated from `knitr::kable()`, e.g., see Table 2.1.

```r
knitr::kable(
  head(iris, 10), caption = 'Here is a nice table!',
  booktabs = TRUE
)
```

You can write citations, too. For example, we are using the **bookdown** package (Xie, 2016) in this sample book, which was built on top of R Markdown and **knitr** (Xie, 2015).

Table 2.1: Here is a nice table!

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa
4.6	3.4	1.4	0.3	setosa
5.0	3.4	1.5	0.2	setosa
4.4	2.9	1.4	0.2	setosa
4.9	3.1	1.5	0.1	setosa

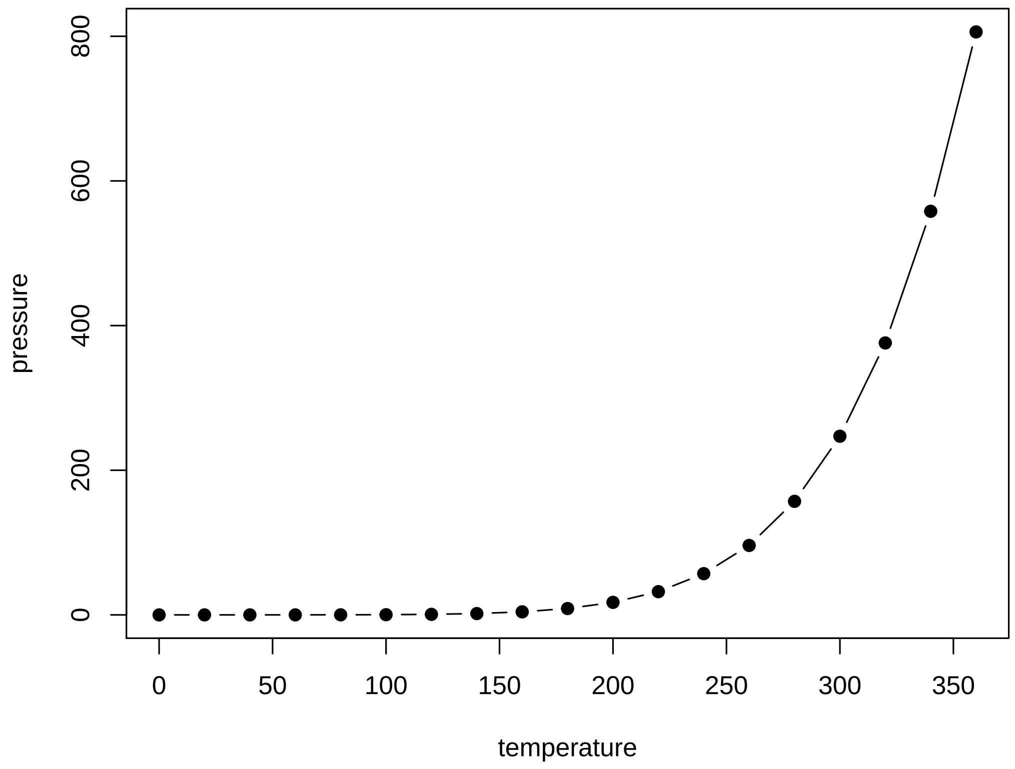


Figure 2.1: Here is a nice figure!

Chapter 3

Multiple Linear Regression

Chapter 4

Logistic Regressions

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

```
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

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

```
##
## 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.

```
anova(model_lr_admission)
```

```
## 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`.

```
library(aod)
wald.test(Sigma = vcov(model_lr_admission), b = coef(model_lr_admission), 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.

```
prediction_lr_admission <- predict(model_lr_admission, data = df2, type = "response")
head(prediction_lr_admission, 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
```

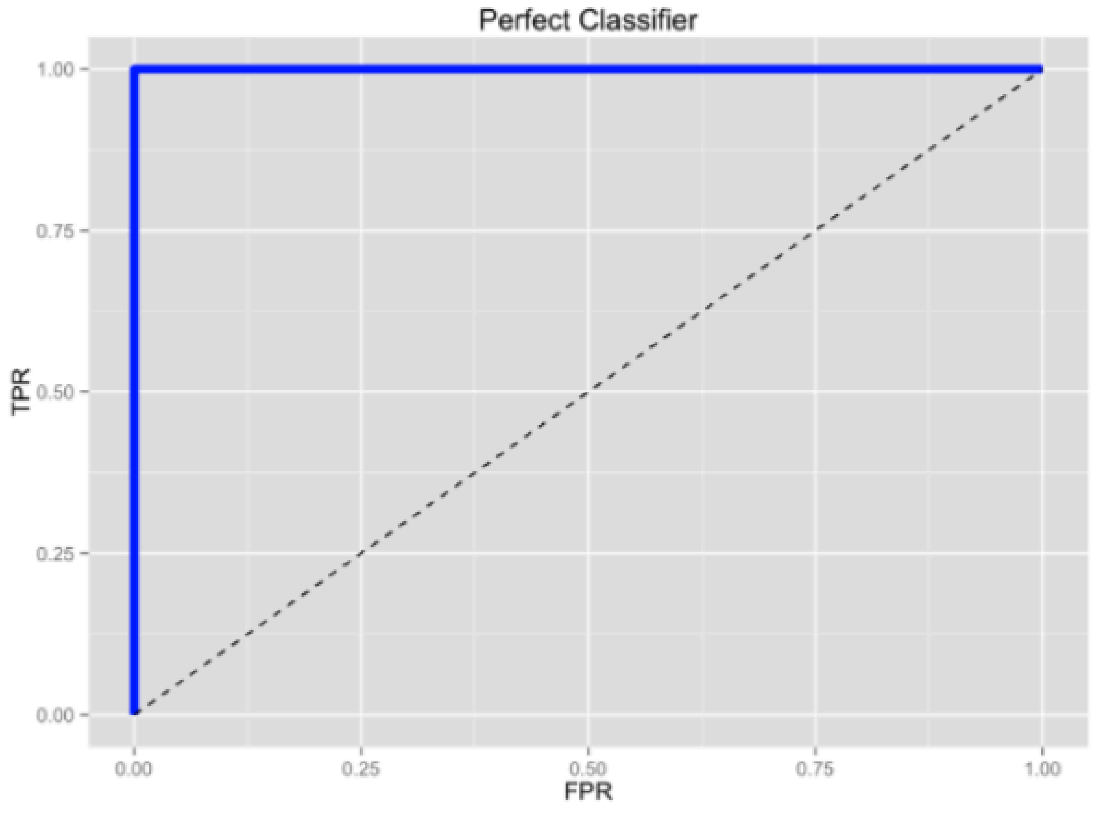
As it stands, the `predict` function gives us the probabilty that the observation has a response of 1; in our case, the probability that a student is being admitted into the graduate program.
To check the accuracy of the model, we need a confusion matrix with a cut off value. So let's clean that vector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

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

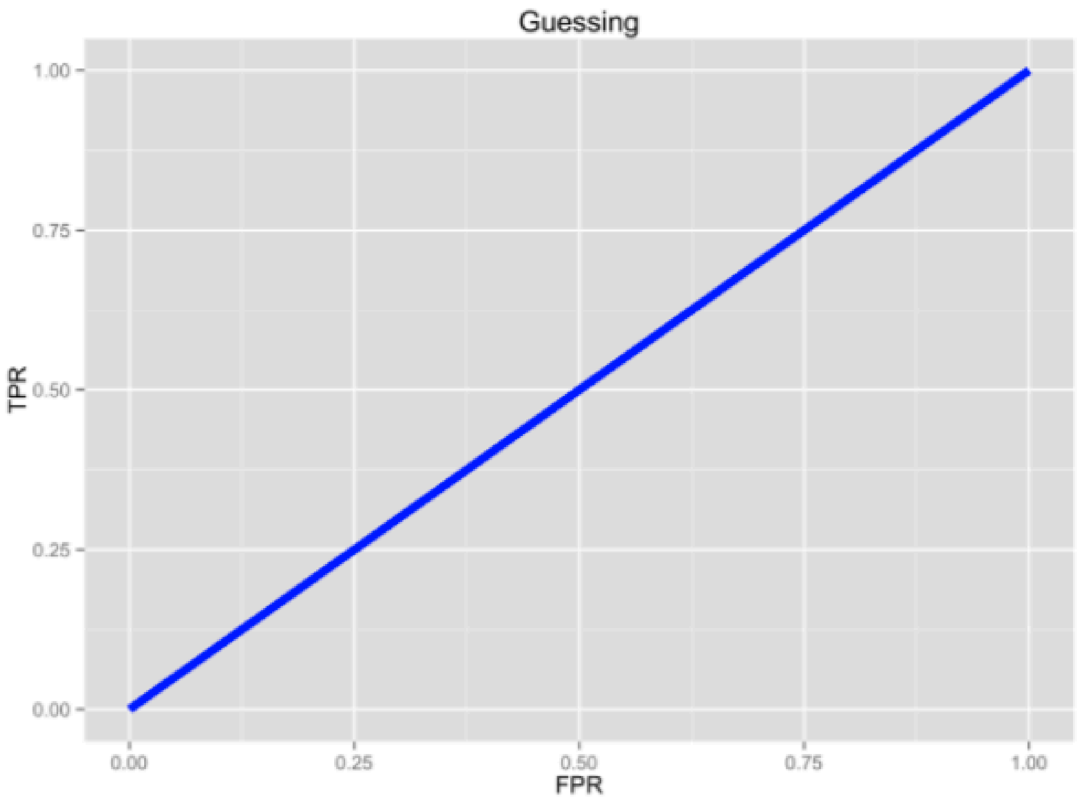
```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 254  97
##          1  19  30
##
##                Accuracy : 0.71
##                  95% CI : (0.6628, 0.754)
##     No Information Rate : 0.6825
##     P-Value [Acc > NIR] : 0.1293
##
##                   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.

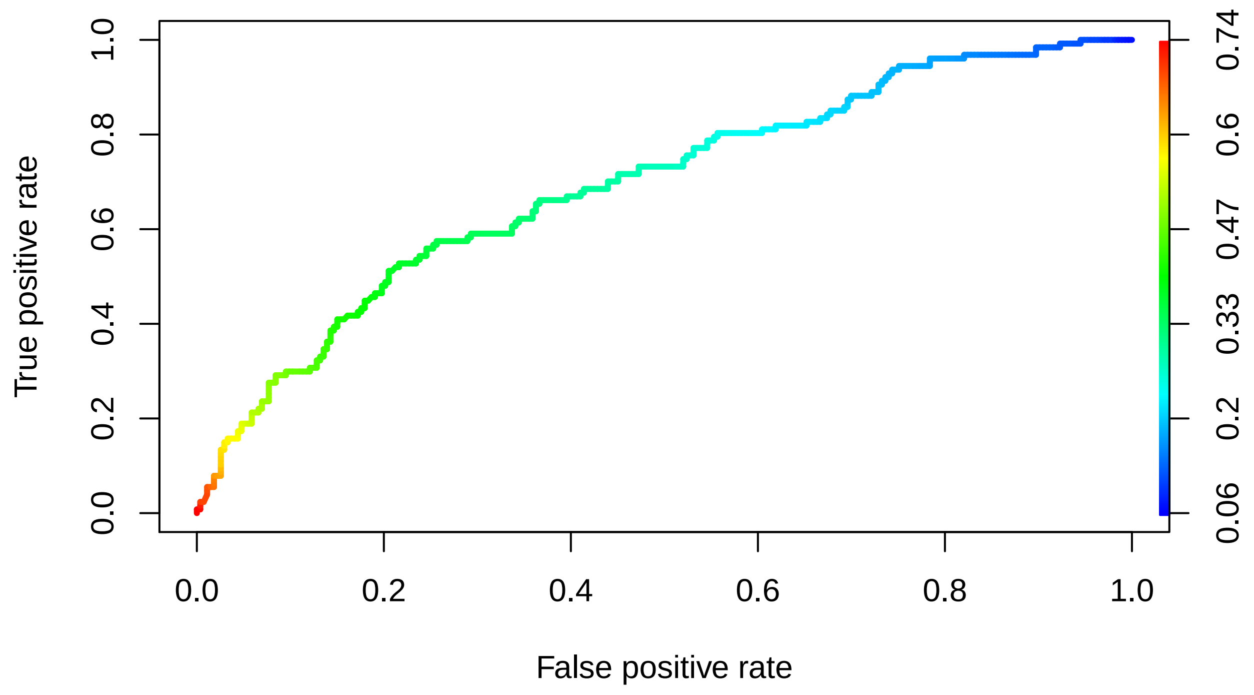


Pure Random guessing should lead to this curve



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

```
prediction_lr_admission <- predict(model_lr_admission, data = df2, type="response")
pr_admission <- prediction(prediction_lr_admission, 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.

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

```
##      392
## 0.487194
```

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

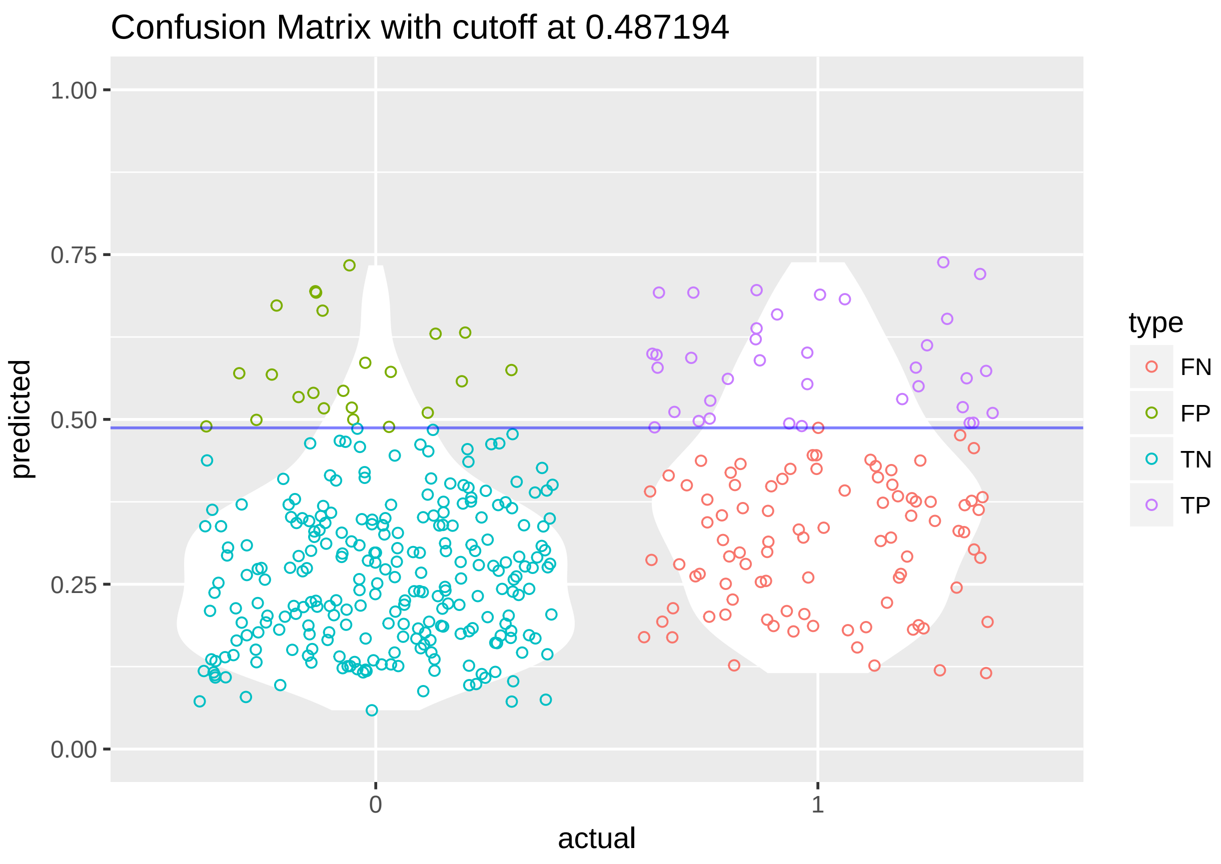
```r
prediction_lr_admission <- predict(model_lr_admission, data = df2, type = "response")
prediction_lr_admission <- if_else(prediction_lr_admission > 0.4721949 , 1, 0)
confusionMatrix(data = prediction_lr_admission,
                reference = df$admit,
                positive = "1")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   0   1
##          0 247  89
##          1  26  38
##
##                Accuracy : 0.7125
##                  95% CI : (0.6654, 0.7564)
##     No Information Rate : 0.6825
##     P-Value [Acc > NIR] : 0.1077
##
##                   Kappa : 0.2352
##  Mcnemar's Test P-Value : 7.402e-09
##
##             Sensitivity : 0.2992
##             Specificity : 0.9048
##          Pos Pred Value : 0.5938
##          Neg Pred Value : 0.7351
##              Prevalence : 0.3175
##          Detection Rate : 0.0950
##    Detection Prevalence : 0.1600
##       Balanced Accuracy : 0.6020
##
##        'Positive' Class : 1
##
```

And bonus, we even gained some accuracy!

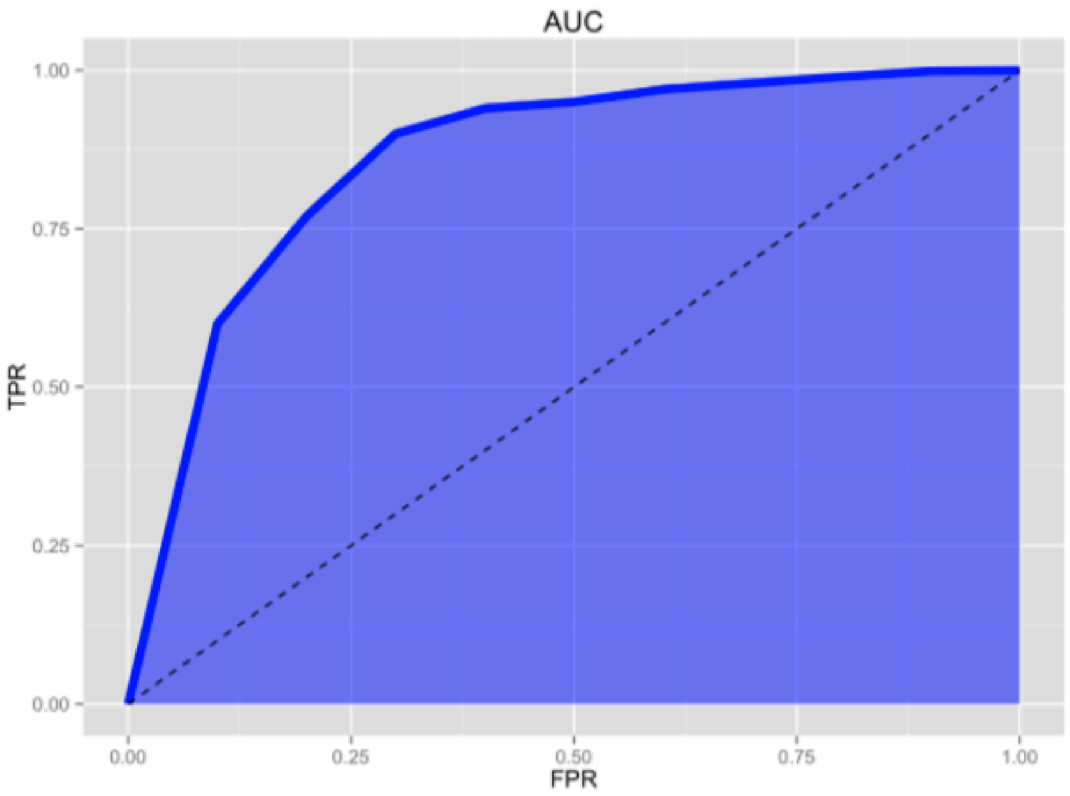
I have seen a very cool graph on this website that plots this tradeoff between specificity and sensitivity and shows how this cutoff point can enhance the understanding of the predictive power of our model.

```r
# Create tibble with both prediction and actual value
cutoff = 0.487194
cutoff_plot <- tibble(predicted = predict(model_lr_admission, 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_lr_admission <- predict(model_lr_admission, data = df2, type="response")
pr_admission <- prediction(prediction_lr_admission, 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

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.

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

```
## 
## Call:
## glm(formula = test ~ ., family = "binomial", data = df)
## 
## 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_diabetes_lr)
```

```
## 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_diabetes_lr))
```

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

table(df$test)
```

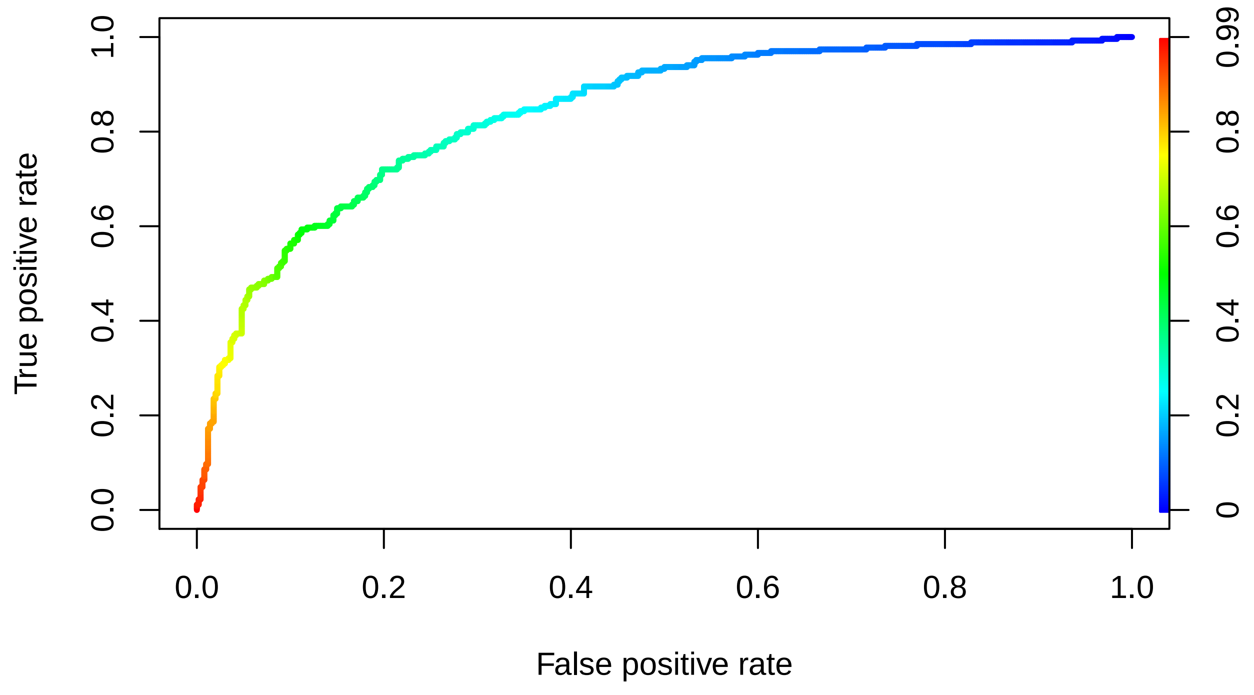
```
##
##   0   1
## 500 268
```

```
confusionMatrix(data = prediction_diabetes_lr,
                reference = df$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

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



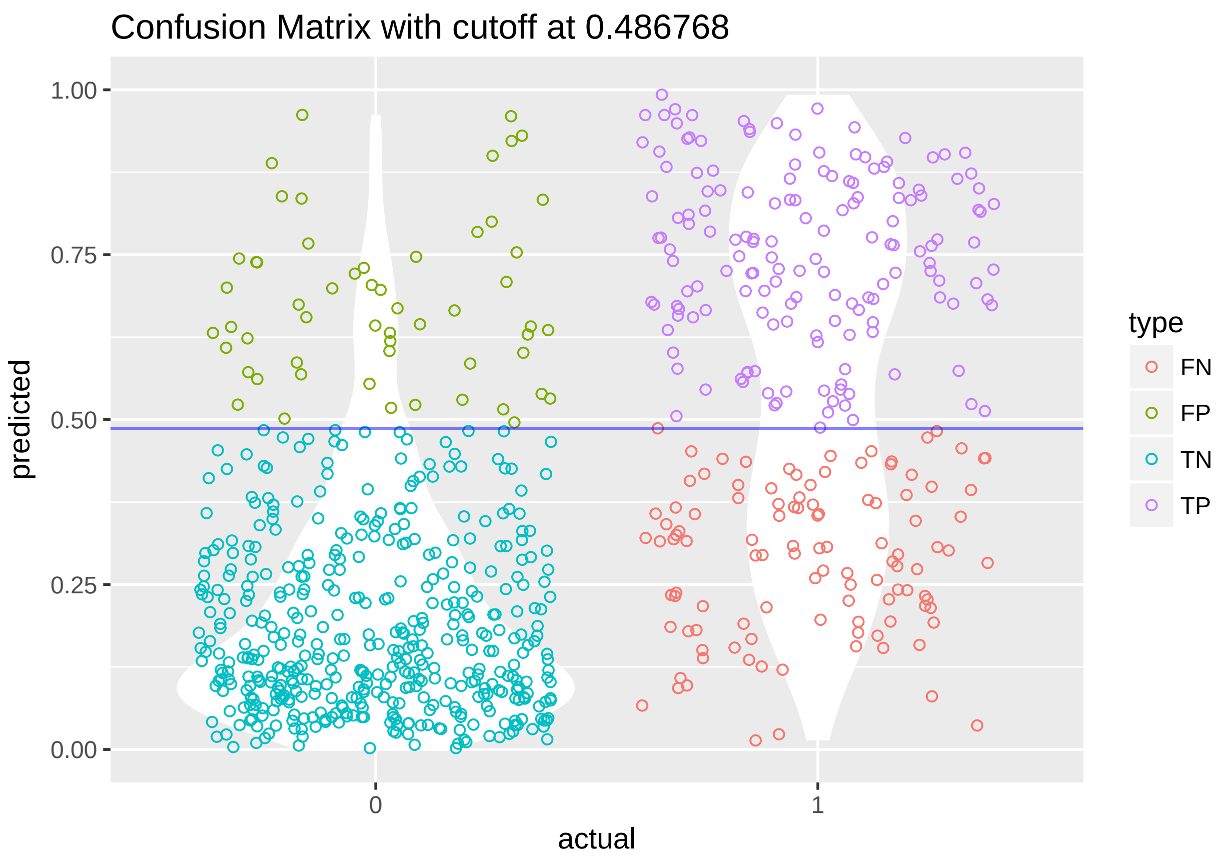
Let's find the best cutoff value for our model.

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

```
##      294
## 0.486768
```

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, "TN", "FN"))))
  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(0.486768, predict(model_diabetes_lr, data = df, type = "response"), df$test)
```



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

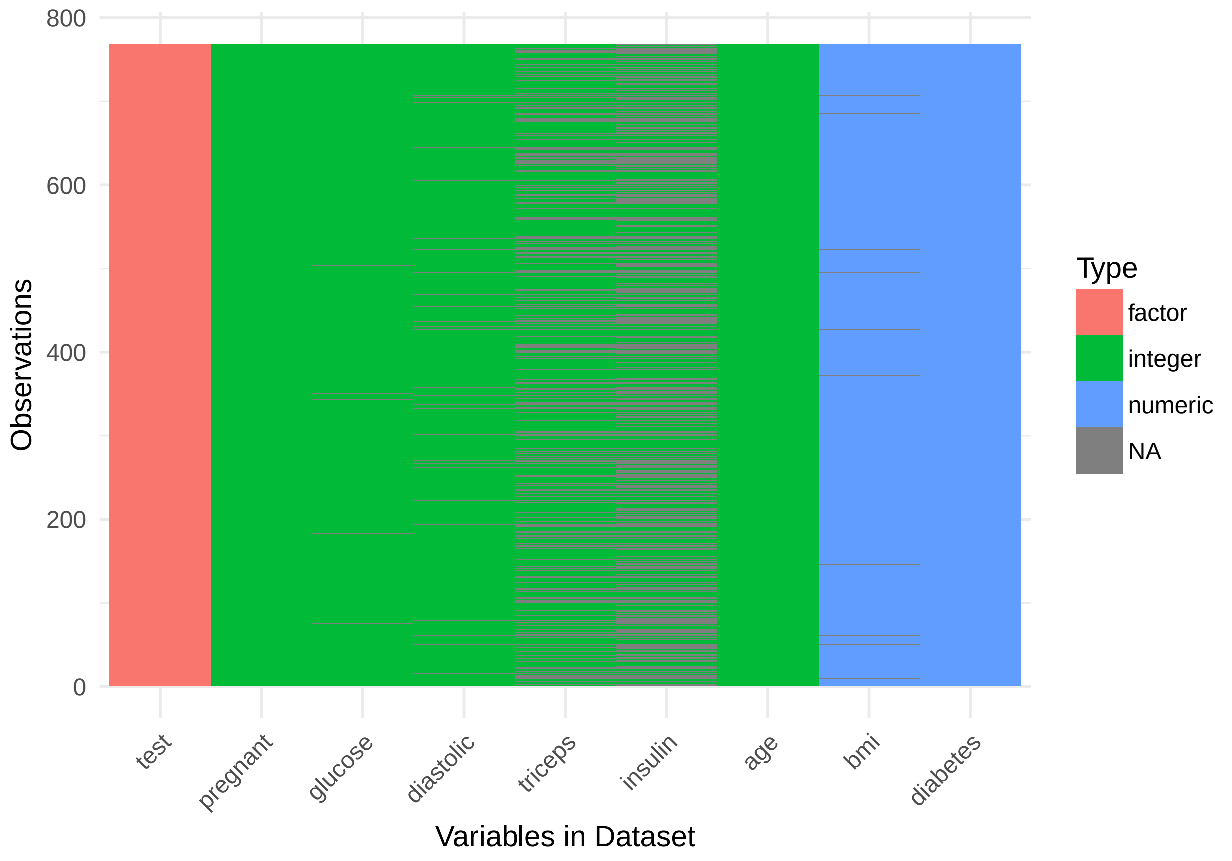
4.6.1 Accounting for missing values

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`.

```
df2 <- df

#TODO Find a way to create a function and use map from purrr to do this
df2$glucose[df2$glucose == 0] <- NA
df2$diastolic[df2$diastolic == 0] <- NA
df2$triceps[df2$triceps == 0] <- NA
df2$insulin[df2$insulin == 0] <- NA
df2$bmi[df2$bmi == 0] <- NA
```

```
library(visdat)
vis_dat(df2)
```



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_diabetes_lr2 <- glm(test ~., data = df2, family = "binomial")
summary(model_diabetes_lr2)
```

```
##
## Call:
## glm(formula = test ~ ., family = "binomial", data = df2)
```

```
##
## 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_diabetes_lr2 <- predict(model_diabetes_lr2, data = df2, type="response")
prediction_diabetes_lr2 <- if_else(prediction_diabetes_lr2 > 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_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)
df3 <- impute_lm(df2, formula = glucose ~ pregnant + diabetes + age | test)
df3 <- impute_rf(df3, formula = bmi ~ glucose + pregnant + diabetes + age | test)
df3 <- impute_rf(df3, formula = diastolic ~ bmi + glucose + pregnant + diabetes + age | test)
df3 <- impute_en(df3, formula = triceps ~ pregnant + bmi + diabetes + age | test)
```

```
df3 <- impute_rf(df3, formula = insulin ~ . | test)

summary(df3)
```

```
##     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 :28.98
##  Mean   : 3.845   Mean   :121.68   Mean   : 72.37   Mean   :28.90
##  3rd Qu.: 6.000   3rd Qu.:141.00   3rd Qu.: 80.00   3rd Qu.:35.00
##  Max.   :17.000   Max.   :199.00   Max.   :122.00   Max.   :99.00
##     insulin            bmi           diabetes           age          test
##  Min.   : 14.00   Min.   :18.20   Min.   :0.0780   Min.   :21.00   0:500
##  1st Qu.: 92.39   1st Qu.:27.50   1st Qu.:0.2437   1st Qu.:24.00   1:268
##  Median :135.00   Median :32.14   Median :0.3725   Median :29.00
##  Mean   :155.64   Mean   :32.43   Mean   :0.4719   Mean   :33.24
##  3rd Qu.:190.59   3rd Qu.:36.60   3rd Qu.:0.6262   3rd Qu.:41.00
##  Max.   :846.00   Max.   :67.10   Max.   :2.4200   Max.   :81.00
```

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

```
model_diabetes_lr3 <- glm(test ~ ., data = df3, family = "binomial")
summary(model_diabetes_lr3)
```

```
##
## Call:
## glm(formula = test ~ ., family = "binomial", data = df3)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -3.2740  -0.7047  -0.3880   0.7130   2.3764
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -8.659337   0.822198 -10.532  < 2e-16 ***
## pregnant     0.127297   0.032256   3.946 7.93e-05 ***
## glucose      0.031118   0.004148   7.502 6.27e-14 ***
## diastolic   -0.006848   0.008684  -0.789   0.4304
## triceps      0.004388   0.014435   0.304   0.7611
## insulin      0.003256   0.001333   2.443   0.0146 *
## bmi          0.081986   0.020812   3.939 8.17e-05 ***
## diabetes     0.836483   0.297978   2.807   0.0050 **
## age          0.009776   0.009708   1.007   0.3139
## ---
## 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: 705.44  on 759  degrees of freedom
## AIC: 723.44
##
## Number of Fisher Scoring iterations: 5
```

```
accuracy_model_lr3 <- predict(model_diabetes_lr3, data = df3, type="response")
accuracy_model_lr3 <- if_else(accuracy_model_lr3 > 0.5, "positive", "negative")
accuracy_model_lr3 <- factor(accuracy_model_lr3)
levels(accuracy_model_lr3) <- c("negative", "positive")



table(df3$test, accuracy_model_lr3)
```

```
##    accuracy_model_lr3
##     negative positive
##   0      442       58
##   1      112      156
```

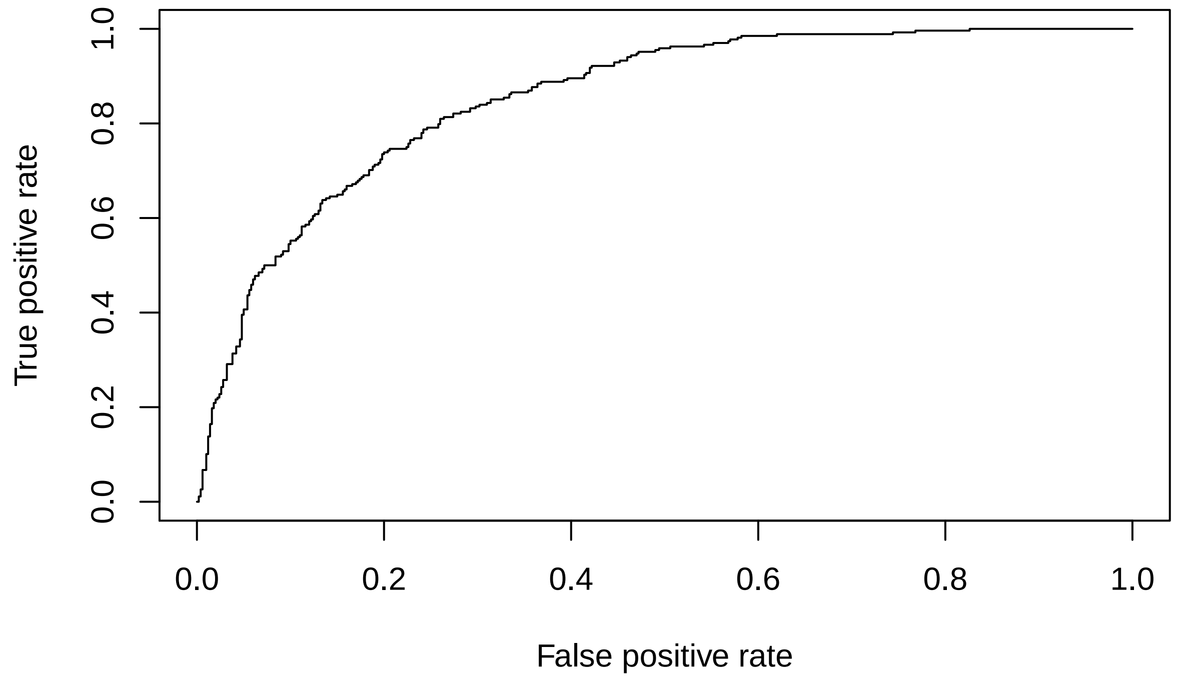
```
table(df3$test)
```

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

```
########

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

4.6.3 ROC and AUC

```
accuracy_model_lr3 <- predict(model_diabetes_lr3, data = df3, type="response")
pr <- prediction(accuracy_model_lr3, df3$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")
pr@cutoffs[[1]][which.min(cost_diabetes_perf@y.values[[1]])]
```

```
##        10
## 0.4525332
```

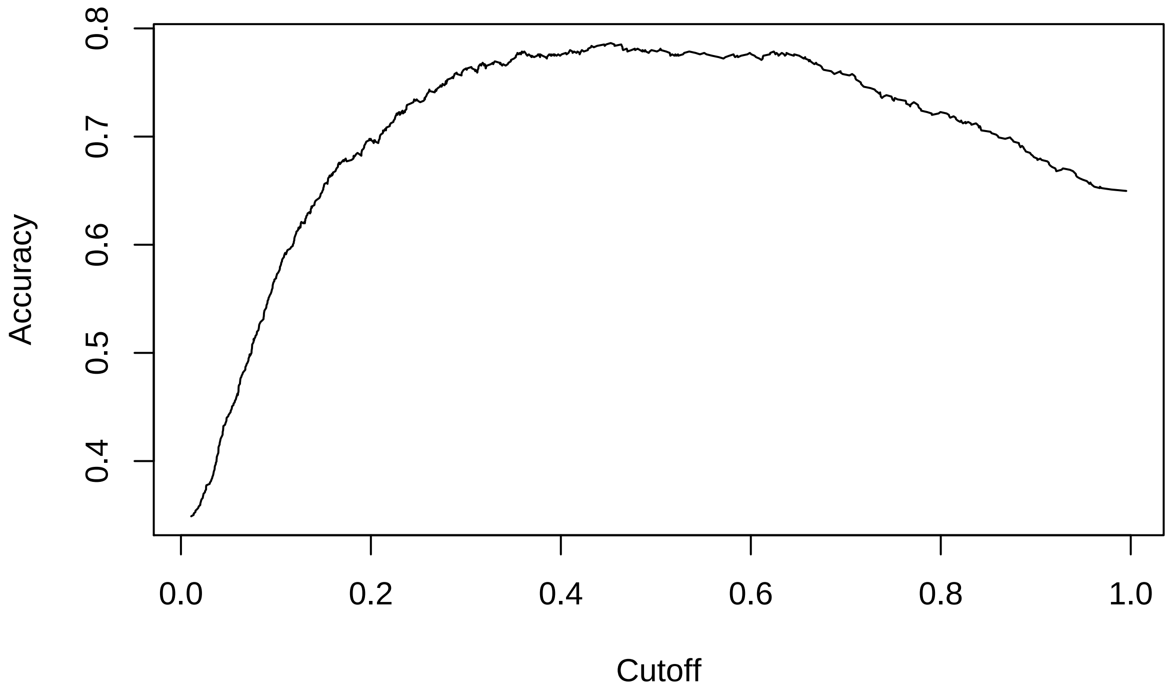
So for maximum accuracy, the ideal cutoff point is `0.487194`.
Let's redo our confusion matrix then and see some improvement.

```
accuracy_model_lr3 <- predict(model_diabetes_lr3, data = df3, type="response")
accuracy_model_lr3 <- if_else(accuracy_model_lr3 >= 0.487194, "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

```
accuracy_diabetes_lr3 <- performance(pr, measure = "acc")
plot(accuracy_diabetes_lr3)
```



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)
pr@cutoffs[[1]][which.min(cost_diabetes_perf@y.values[[1]])]
```

```
##       477
## 0.2271368
```

Lastly, in regards to AUC

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

```
## [1] 0.8530075
```

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

Trees and Classification

5.1 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.

5.2 First example.

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

```r
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.

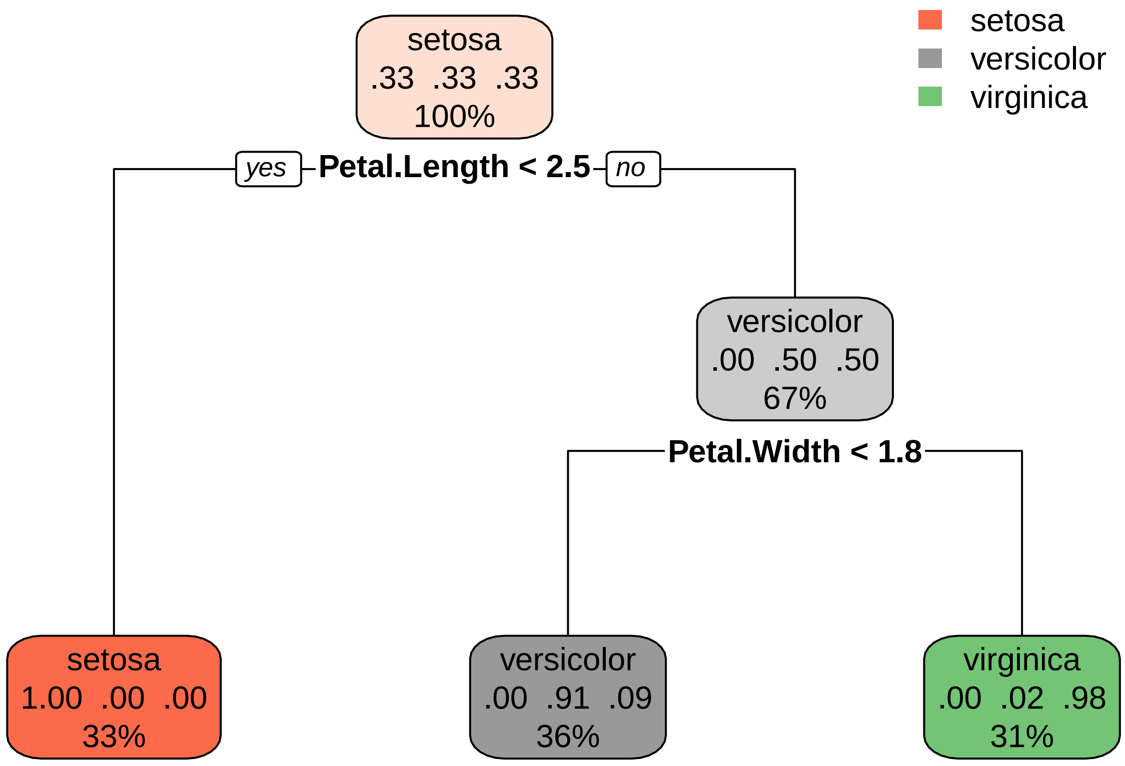
Leaf/ Terminal Node: Nodes do not split is called Leaf or Terminal node.

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.

```
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

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

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

5.3 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)
```

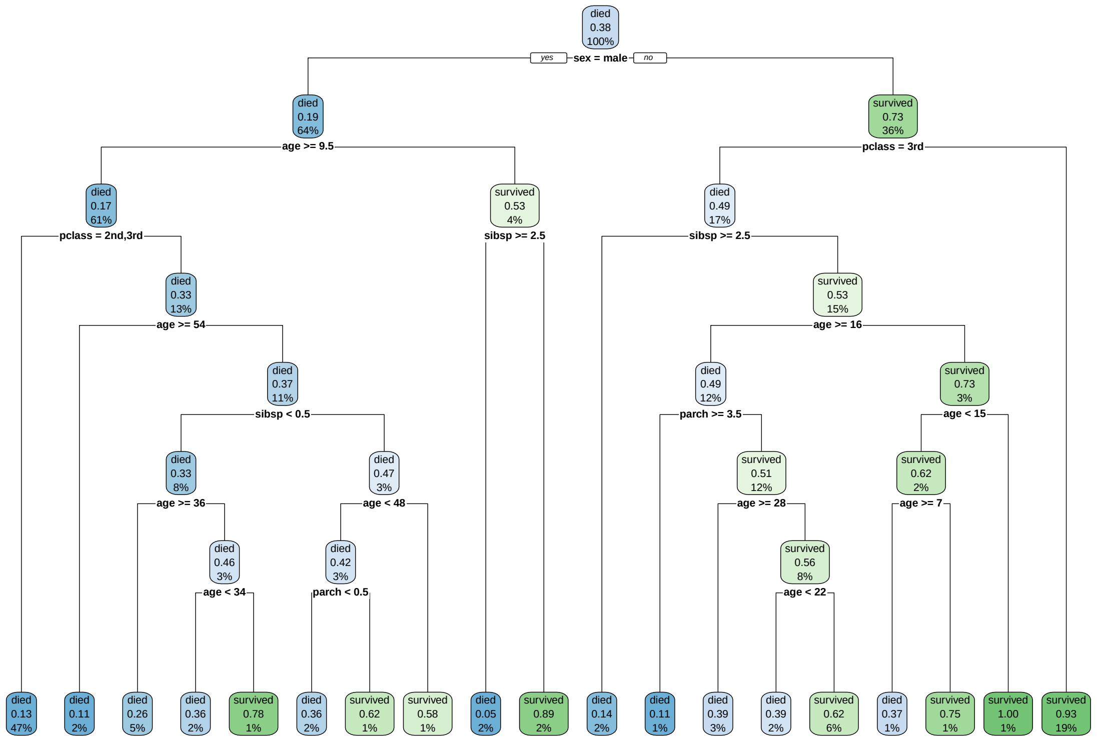
```
##
##          died survived
##   female 0.27     0.73
##   male   0.81     0.19
```

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")), 2), 2)
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.

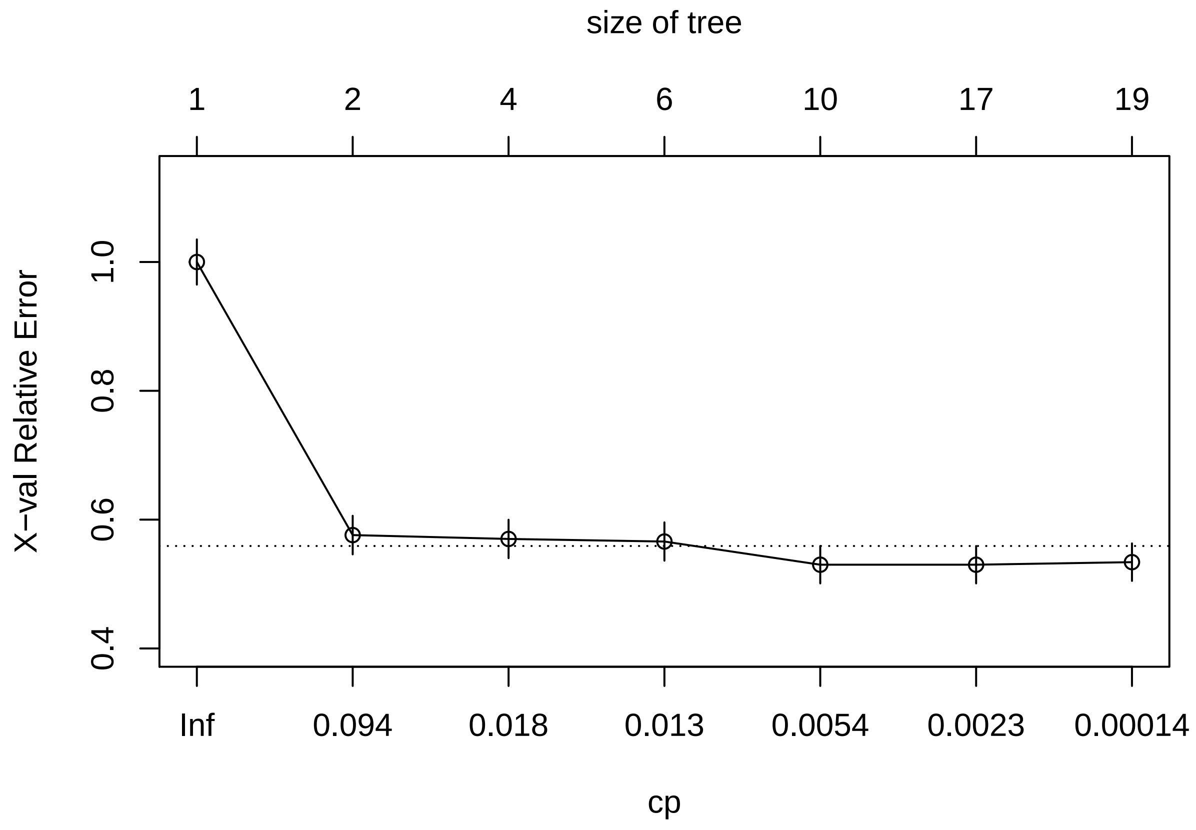
As discussed earlier, the technique of setting constraint is a greedy-approach. In other words, it will check for the best split instantaneously and move forward until one of the specified stopping condition is reached. Let's consider the following case when you're driving: There are 2 lanes: A lane 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
```

```r
plotcp(tree)
```

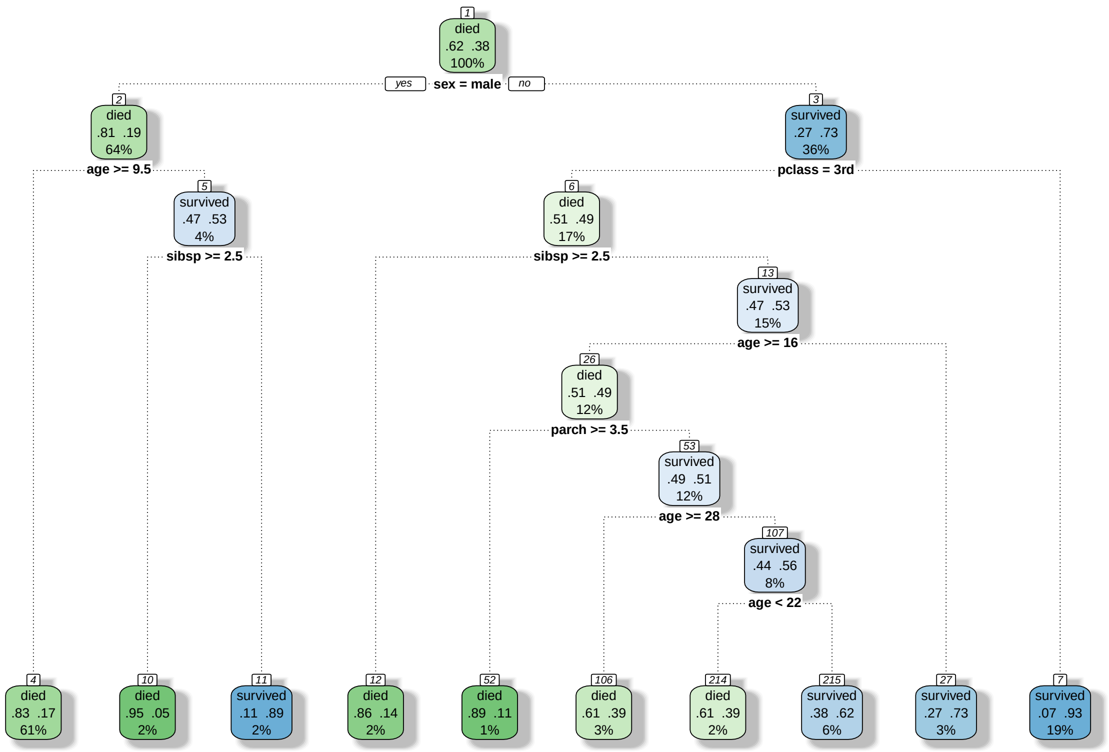


```r
tree$cptable[which.min(tree$cptable[,"xerror"]),"CP"]
```

```
## [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 graph!)
rpart.plot(tree.pruned, extra=104, box.palette="GnBu",
               branch.lty=3, shadow.col="gray", nn=TRUE)
```

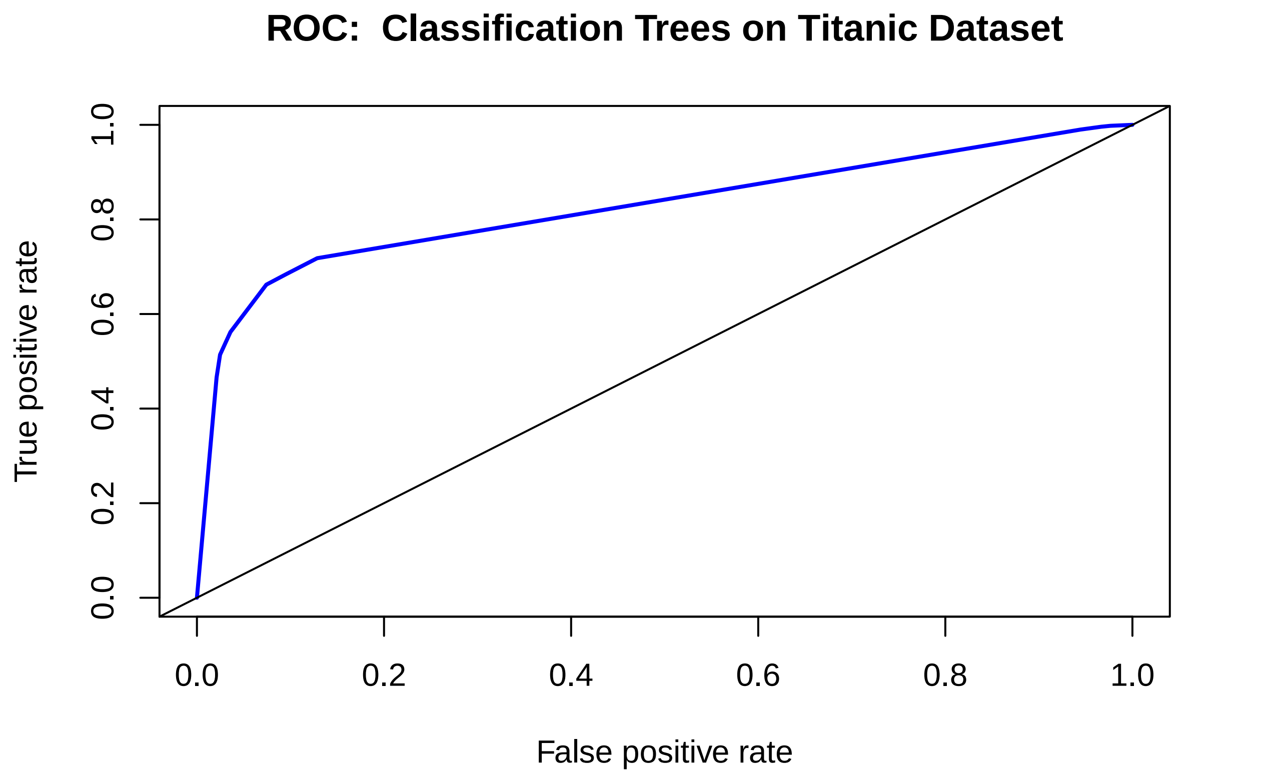


```
conf.matrix <- round(prop.table(table(ptitanic$survived, predict(tree.pruned, type="class"))), 2)
rownames(conf.matrix) <- c("Actually died", "Actually Survived")
colnames(conf.matrix) <- c("Predicted dead", "Predicted Survived")
conf.matrix
```

```
##
##                     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.

5.4 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.

5.5 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 # 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.

5.6 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 6

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.

6.1 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"
```

```
print(pca_buycomputer)
```

```
## Standard deviations:
## [1] 1.5589391 0.9804092 0.6816673 0.3792578
##
## Rotation:
##                   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
```

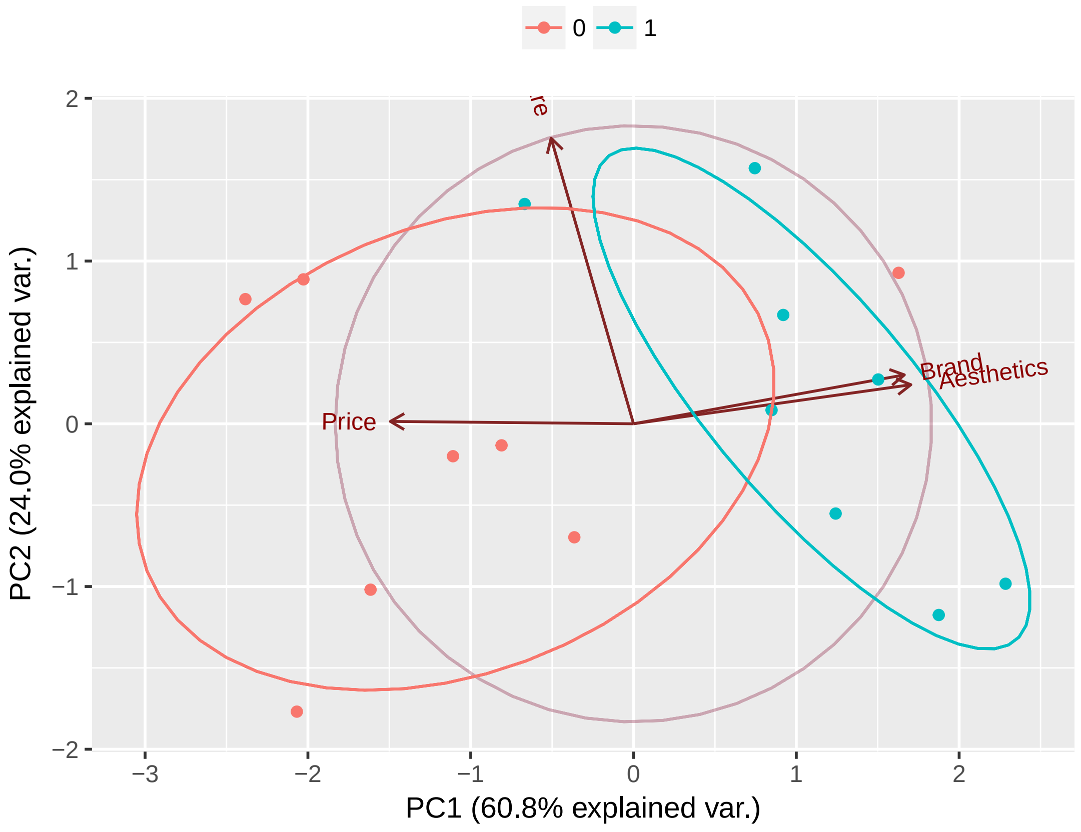
```
summary(pca_buycomputer, loadings = TRUE)
```

```
## Warning: In summary.prcomp(pca_buycomputer, loadings = TRUE) :
##  extra argument 'loadings' will be disregarded
```

```
## 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
```

```
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.

```r
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.

6.2 Attempt of PCA on technical indicators.

For this purpose, we have taken a random stock, added a lots of variables and have one dependent variable.

```r
# Read the file
library(readr)
stock_data <- read_csv("AugmentedStockData/CVX.csv")
```

Now onto create our dependent variable and stipping down the data frame to just the columns that interest us, and only get rows and columns without NA.

```r
library(dplyr)
binary = if_else(stock_data$ret3days[25:4150] > 0.03, 1, 0)
depend_var <- stock_data[25:4150,8:34]
```

The base R function `prcomp()` is used to perform PCA. PCA only works with normalized data. So we need to center the variable to have mean equals to zero. With parameter `scale. = T`, we normalize the variables to have standard deviation equals to 1. **Normalized predictors have mean equals to zero and standard deviation equals to one.**

```
prin_comp <- prcomp(depend_var, scale. = TRUE, center = TRUE)
```

Let's have a closer look at that 'prcomp' function.

`Center` and `scale` refers to respective mean and standard deviation of the variables that are used for normalization prior to implementing PCA.

```
names(prin_comp)
```

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

```
summary(prin_comp)
```

```
## Importance of components:
##                           PC1    PC2    PC3     PC4     PC5     PC6
## Standard deviation     3.3007 2.5311 1.6647 1.58184 1.12216 0.89194
## Proportion of Variance 0.4035 0.2373 0.1026 0.09268 0.04664 0.02947
## Cumulative Proportion  0.4035 0.6408 0.7434 0.83609 0.88273 0.91220
##                            PC7     PC8     PC9    PC10    PC11    PC12
## Standard deviation     0.80601 0.63478 0.59745 0.53196 0.42057 0.39748
## Proportion of Variance 0.02406 0.01492 0.01322 0.01048 0.00655 0.00585
## Cumulative Proportion  0.93626 0.95118 0.96440 0.97488 0.98143 0.98728
##                           PC13    PC14    PC15    PC16    PC17    PC18
## Standard deviation     0.36869 0.33376 0.18342 0.17136 0.14502 0.08488
## Proportion of Variance 0.00503 0.00413 0.00125 0.00109 0.00078 0.00027
## Cumulative Proportion  0.99232 0.99644 0.99769 0.99878 0.99956 0.99982
##                           PC19    PC20    PC21    PC22     PC23     PC24
## Standard deviation     0.05953 0.02347 0.01988 0.01569 0.003083 0.002066
## Proportion of Variance 0.00013 0.00002 0.00001 0.00001 0.000000 0.000000
## Cumulative Proportion  0.99996 0.99998 0.99999 1.00000 1.000000 1.000000
##                             PC25      PC26      PC27
## Standard deviation     2.065e-15 1.629e-15 1.194e-15
## Proportion of Variance 0.000e+00 0.000e+00 0.000e+00
## Cumulative Proportion  1.000e+00 1.000e+00 1.000e+00
```

```
#outputs the mean of variables
prin_comp$center
```

```
##         wma3         wma5         wma7         wma9        wma11
## 2.621433e-05 7.849430e-05 1.321253e-04 1.852866e-04 2.367136e-04
##     rsi_3val     rsi_3dir     rsi_5val     rsi_5dir     rsi_7val
## 5.249290e-01 2.259665e-01 5.219632e-01 5.112341e-02 5.198979e-01
##     rsi_7dir     rsi_9val     rsi_9dir    rsi_11val    rsi_11dir
## 2.280991e-02 5.183530e-01 1.295840e-02 5.171323e-01 8.388109e-03
##       11arup      11ardow     11arosci       19arup      19ardow
## 5.318600e-01 4.556251e-01 7.623496e-02 5.355895e-01 4.540398e-01
##     19arosci       23arup      23ardow     23arosci ave_vol3days
## 8.154961e-02 5.418344e-01 4.505785e-01 9.125587e-02 1.393280e-03
## ave_vol5days ave_vol7days
## 3.236529e-03 4.388667e-03
```

```
#outputs the standard deviation of variables
prin_comp$scale
```

```
##         wma3         wma5         wma7         wma9        wma11
##   0.01014433   0.01488609   0.01842084   0.02129883   0.02378999
##     rsi_3val     rsi_3dir     rsi_5val     rsi_5dir     rsi_7val
##   0.25196379   1.55382629   0.19466505   0.40810579   0.16395437
##     rsi_7dir     rsi_9val     rsi_9dir    rsi_11val    rsi_11dir
##   0.23886699   0.14412998   0.17015563   0.13002831   0.13287362
##       11arup      11ardow     11arosci       19arup      19ardow
##   0.37456507   0.36908431   0.65809034   0.36982907   0.36010315
##     19arosci       23arup      23ardow     23arosci ave_vol3days
##   0.64812635   0.36425219   0.35741487   0.63505784   0.18659298
## ave_vol5days ave_vol7days
##   0.22564569   0.24793982
```

The rotation measure provides the principal component loading. Each column of rotation matrix contains the principal component loading vector. This is the most important measure we should be interested in.

```
#because it can be a huge matrix, let's only check the first few rows and columns.
prin_comp$rotation[1:5, 1:5]
```

```
##             PC1       PC2         PC3         PC4       PC5
## wma3  0.1788016 0.2543854 -0.00981180 -0.08087736 0.2431686
## wma5  0.2086344 0.2209183  0.02753808 -0.18315739 0.2465858
## wma7  0.2252186 0.1839885  0.04507260 -0.23309018 0.2168579
## wma9  0.2360512 0.1499467  0.05369872 -0.25316224 0.1770487
## wma11 0.2437448 0.1198431  0.05604151 -0.25599437 0.1376221
```

Let's plot the resultant principal components.

The prcomp() function also provides the facility to compute standard deviation of each principal component. sdev refers to the standard deviation of principal components.

```
#compute standard deviation of each principal component
std_dev <- prin_comp$sdev

#compute variance
pr_var <- std_dev^2

#check variance of first 10 components
pr_var[1:10]
```

```
##  [1] 10.8943784  6.4065586  2.7713407  2.5022255  1.2592331  0.7955608
##  [7]  0.6496465  0.4029457  0.3569417  0.2829776
```

We aim to find the components which explain the maximum variance. This is because, we want to retain as much information as possible using these components. So, higher is the explained variance, higher will be the information contained in those components.

To compute the proportion of variance explained by each component, we simply divide the variance by sum of total variance. This results in:

```
#proportion of variance explained
prop_varex <- pr_var/sum(pr_var)
prop_varex[1:20]
```

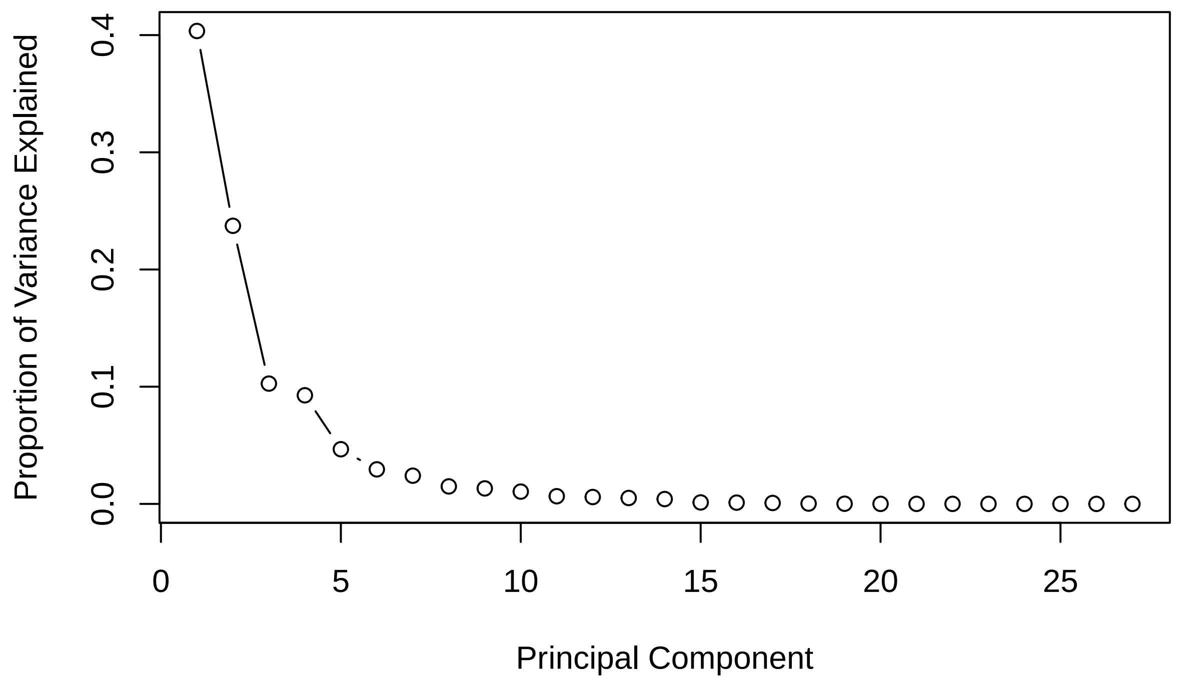
```
##  [1] 4.034955e-01 2.372799e-01 1.026422e-01 9.267502e-02 4.663826e-02
##  [6] 2.946522e-02 2.406098e-02 1.492391e-02 1.322006e-02 1.048065e-02
## [11] 6.550995e-03 5.851611e-03 5.034607e-03 4.125722e-03 1.246026e-03
## [16] 1.087586e-03 7.789197e-04 2.668137e-04 1.312590e-04 2.039572e-05
```

This shows that first principal component explains 41.7% variance. Second component explains 23.8%

variance. Third component explains 10.4% variance and so on. So, how do we decide how many components should we select for modeling stage ?

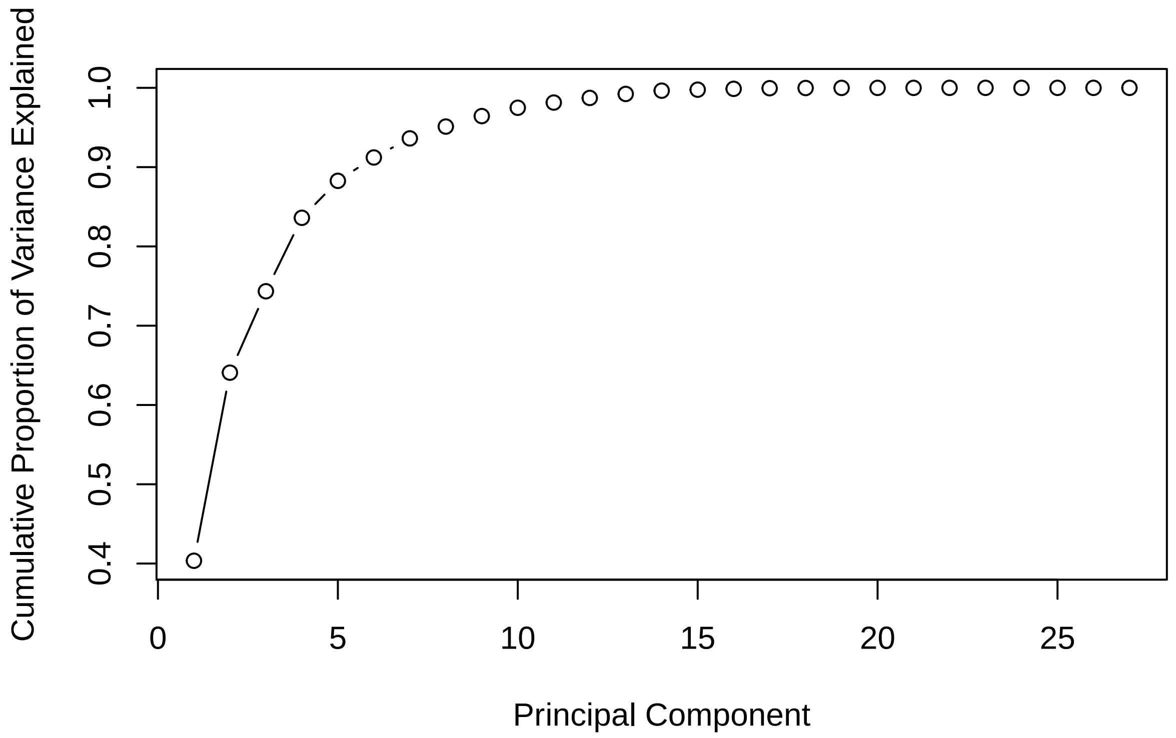
The answer to this question is provided by a scree plot. A scree plot is used to access components or factors which explains the most of variability in the data. It represents values in descending order.

```r
#scree plot
plot(prop_varex, xlab = "Principal Component",
             ylab = "Proportion of Variance Explained",
             type = "b")
```



Or we can do a cumulative scree plot

```r
#cumulative scree plot
plot(cumsum(prop_varex), xlab = "Principal Component",
              ylab = "Cumulative Proportion of Variance Explained",
              type = "b")
```



Hence in this case the first 6 Principal Components explain over 90% of the variance of the data. That is we can use these first 6 PC as predictor in our next model.

Let's apply this now on a logisitc regression model. For this, we need to create our binary dependent variable. So we'll put a 1 for every ret3days > 3%, 0 otherwise.

```
mydata <- cbind(binary, prin_comp$x)
mydata <- as.data.frame(mydata)
model1 <- glm(binary ~ PC1 + PC2 + PC5 + PC6 + PC7, data = mydata, family=binomial)
summary(model1)
```

```
##
## Call:
## glm(formula = binary ~ PC1 + PC2 + PC5 + PC6 + PC7, family = binomial,
##     data = mydata)
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -1.5159  -0.5197  -0.4511  -0.3740   2.4644
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) -2.15924    0.05268 -40.989  < 2e-16 ***
## PC1         -0.11636    0.01514  -7.685 1.53e-14 ***
## PC2          0.03253    0.01815   1.793   0.0731 .
## PC5         -0.05541    0.04094  -1.353   0.1760
## PC6         -0.08353    0.05546  -1.506   0.1320
## PC7         -0.04292    0.05665  -0.758   0.4486
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
```

```
##
##     Null deviance: 2851.7  on 4125  degrees of freedom
## Residual deviance: 2784.3  on 4120  degrees of freedom
## AIC: 2796.3
##
## Number of Fisher Scoring iterations: 5
```

```
mydata <- cbind(binary, prin_comp$x)
mydata <- as.data.frame(mydata)
checkit <- fitted(model1)
checkit <- cbind(binary, checkit)
checkit <- as.data.frame(checkit)
head(checkit %>% filter(binary == 1), 20)
```

```
##    binary    checkit
## 1       1 0.18260120
## 2       1 0.18252003
## 3       1 0.21606180
## 4       1 0.21251118
## 5       1 0.11869114
## 6       1 0.14243678
## 7       1 0.15250266
## 8       1 0.08137055
## 9       1 0.09081918
## 10      1 0.07178808
## 11      1 0.08675686
## 12      1 0.08601504
## 13      1 0.05538413
## 14      1 0.13325771
## 15      1 0.11428528
## 16      1 0.13216528
## 17      1 0.08095967
## 18      1 0.07966423
## 19      1 0.09559465
## 20      1 0.16822463
```

```
head(checkit %>% filter(checkit > 0.2), 20)
```

```
##    binary   checkit
## 1       0 0.2199392
## 2       1 0.2160618
## 3       1 0.2125112
## 4       0 0.2136409
## 5       1 0.2187997
## 6       1 0.2116685
## 7       0 0.2070526
## 8       0 0.2038604
## 9       0 0.2097841
## 10      0 0.2034205
## 11      0 0.2187273
## 12      1 0.2275500
## 13      1 0.2281150
## 14      0 0.2183669
## 15      0 0.2060439
## 16      0 0.2201554
## 17      0 0.2692442
## 18      1 0.2772148
## 19      1 0.2574167
## 20      1 0.2005783
```

Really not very successful model.

6.3 Doing PCA and PCR with the PLS package

Same as before we can not have NA data in our set.

```
library(pls)
```

```
##
## Attaching package: 'pls'

## The following object is masked from 'package:caret':
##
##     R2

## The following object is masked from 'package:stats':
##
##     loadings
```

```
depend_var <- stock_data[25:4150,8:35]
pcr_model <- pcr(ret3days~., data = depend_var, scale = TRUE, validation = "CV")
```

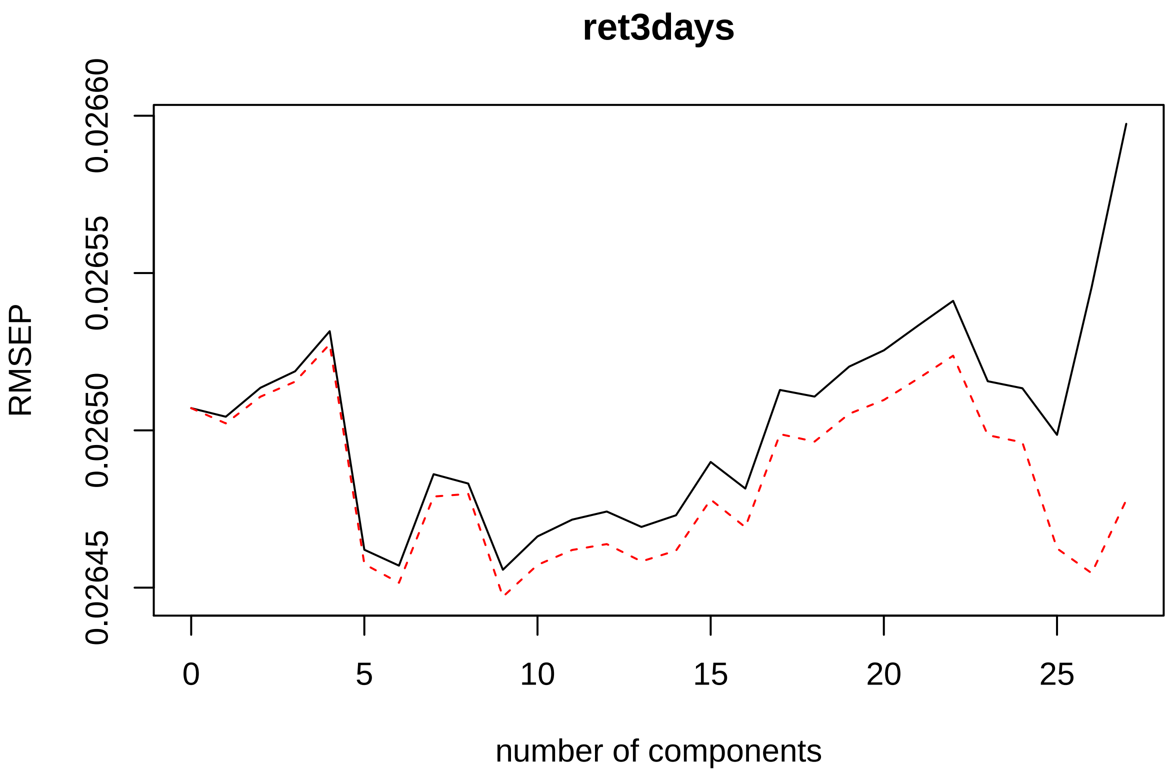
In oder to print out the results, simply use the summary function as below

```
summary(pcr_model)
```

```
## Data:    X dimension: 4126 27
##  Y dimension: 4126 1
## Fit method: svdpc
## Number of components considered: 27
##
## VALIDATION: RMSEP
## Cross-validated using 10 random segments.
##        (Intercept)  1 comps  2 comps  3 comps  4 comps  5 comps  6 comps
## CV         0.02651   0.0265  0.02651  0.02652  0.02653  0.02646  0.02646
## adjCV      0.02651   0.0265  0.02651  0.02652  0.02653  0.02646  0.02645
##        7 comps  8 comps  9 comps  10 comps  11 comps  12 comps  13 comps
## CV     0.02649  0.02648  0.02646   0.02647   0.02647   0.02647   0.02647
## adjCV  0.02648  0.02648  0.02645   0.02646   0.02646   0.02646   0.02646
##        14 comps  15 comps  16 comps  17 comps  18 comps  19 comps
## CV      0.02647   0.02649   0.02648   0.02651   0.02651   0.02652
## adjCV   0.02646   0.02648   0.02647   0.02650   0.02650   0.02651
##        20 comps  21 comps  22 comps  23 comps  24 comps  25 comps
## CV      0.02653   0.02653   0.02654   0.02652   0.02651   0.02650
## adjCV   0.02651   0.02652   0.02652   0.02650   0.02650   0.02646
##        26 comps  27 comps
## CV      0.02655   0.02660
## adjCV   0.02645   0.02648
##
## TRAINING: % variance explained
##           1 comps  2 comps  3 comps  4 comps  5 comps  6 comps  7 comps
## X          40.3495  64.0775  74.3418   83.609  88.2731    91.22   93.626
## ret3days    0.2797   0.3122   0.3381    0.359   0.9058     1.09    1.103
##           8 comps  9 comps  10 comps  11 comps  12 comps  13 comps
## X          95.118   96.440    97.488     98.14    98.728    99.232
## ret3days    1.122    1.521     1.523      1.56     1.685     1.738
##           14 comps  15 comps  16 comps  17 comps  18 comps  19 comps
## X           99.644    99.769    99.878    99.956    99.982    99.996
## ret3days     1.745     1.746     1.796     1.797     1.847     1.865
##           20 comps  21 comps  22 comps  23 comps  24 comps  25 comps
## X            100.0    99.999   100.000   100.000   100.000   100.000
## ret3days       1.9     1.997     2.017     2.166     2.215     2.268
##           26 comps  27 comps
## X          100.000   100.000
## ret3days     2.594     2.632
```

As you can see, two main results are printed, namely the **validation error** and the **cumulative percentage of variance** explained using n components. The cross validation results are computed for each number of components used so that you can easily check the score with a particular number of components without trying each combination on your own. The pls package provides also a set of methods to plot the results of PCR. For example you can plot the results of cross validation using the `validationplot` function. By default, the pcr function computes the **root mean squared error** and the `validationplot` function plots this statistic, however you can choose to plot the usual mean squared error or the R2 by setting the val.type argument equal to “MSEP” or “R2” respectively

```
# Plot the root mean squared error
validationplot(pcr_model)
```



What you would like to see is a low cross validation error with a lower number of components than the number of variables in your dataset. If this is not the case or if the smalles cross validation error occurs with a number of components close to the number of variables in the original data, then no dimensionality reduction occurs. In the example above, it looks like 3 components are enough to explain more than 90% of the variability in the data. Now you can try to use PCR on a traning-test set and evaluate its performance using, for example, using only 6 components

```
# Train-test split
train <- stock_data[25:3000,8:35]
y_test <- stock_data[3001:4150,35]
test <- stock_data[3001:4150,8:34]

pcr_model <- pcr(ret3days~., data = train,scale =TRUE, validation = "CV")

pcr_pred <- predict(pcr_model, test, ncomp = 6)
mean((pcr_pred - y_test)^2)
```

```
## [1] 0.0005600123
```

6.4 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 7

Case Study - Mushrooms Classification

This chapter demonstrates how to classify muhsrooms as edible or not. It also answera 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.

7.1 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 file
library(readr)
library(dplyr)

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.

7.2 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_color_above_ring
                        "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
library(purrr)
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", "pungent", "spicy",
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", "orange",
                                 "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$stalk_color_above_ring) <- c("buff", "cinnamon", "red", "gray", "brown", "pink",
                                "green", "purple", "white", "yellow")
levels(mushroom$stalk_color_below_ring) <- c("buff", "cinnamon", "red", "gray", "brown", "pink",
                                "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", "several", "solitary
levels(mushroom$habitat) <- c("wood", "grasses", "leaves", "meadows", "paths", "urban", "waste")


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...
```

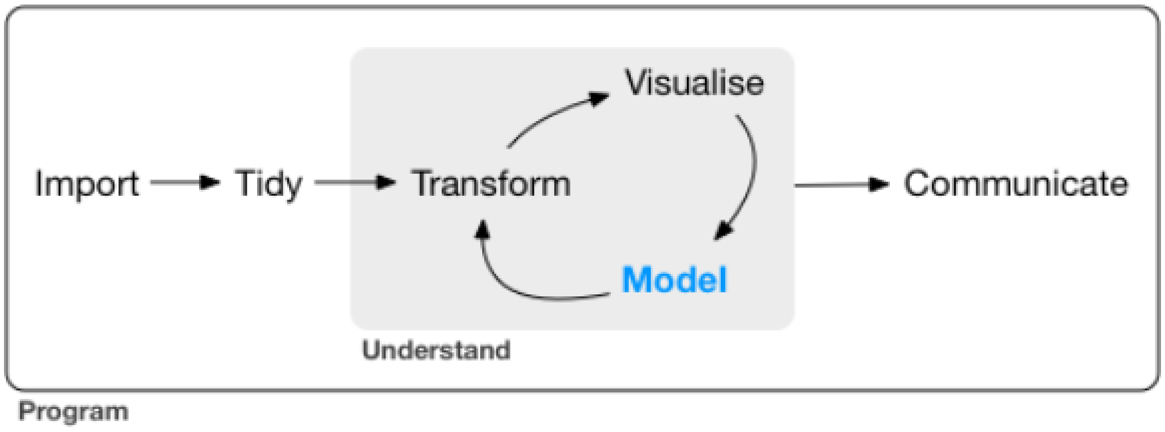
As each variables is categorical, let's see how many categories are we speaking about?

```r
library(tibble)
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 × 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
```

7.3 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.



7.3.1 A. 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))})
```

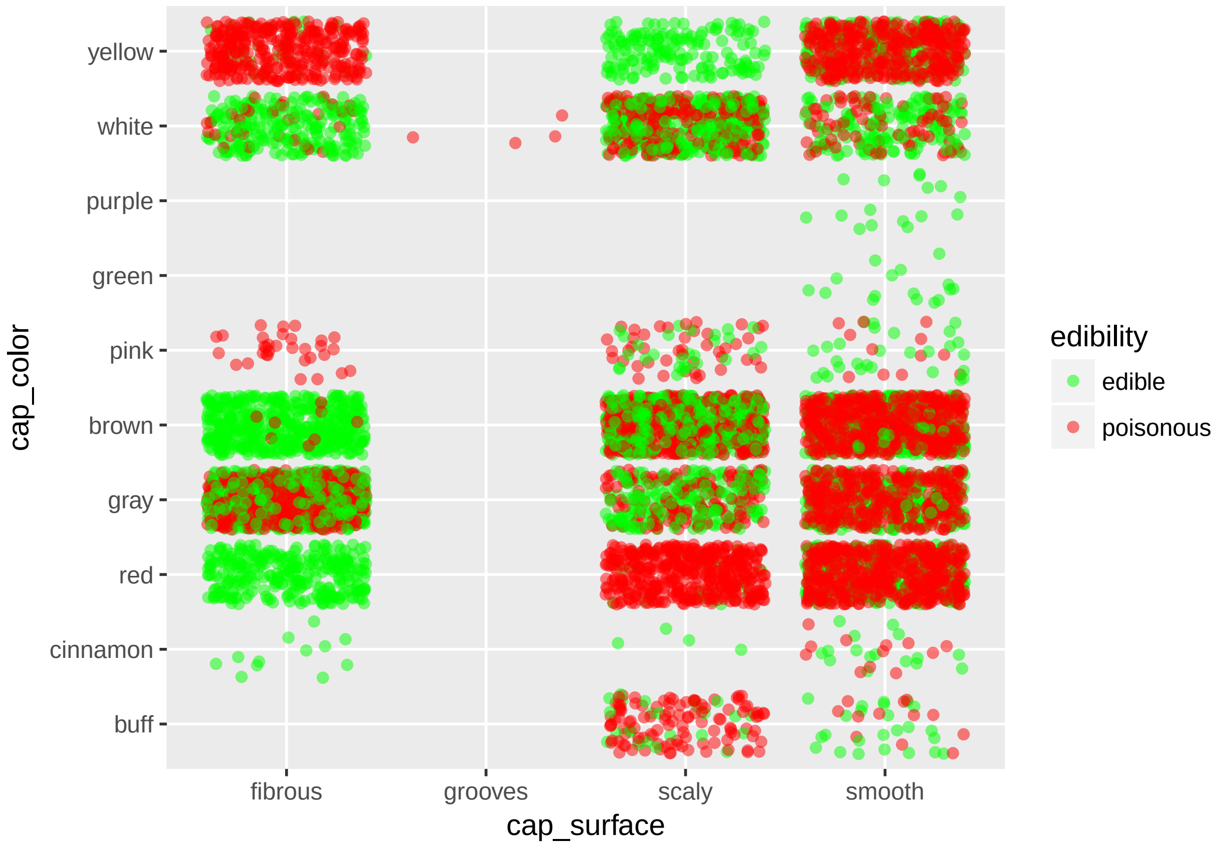
```
##               edibility                cap_shape              cap_surface
##                       0                        0                        0
##               cap_color                  bruises                     odor
##                       0                        0                        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.

7.3.2 A. Visualize the data

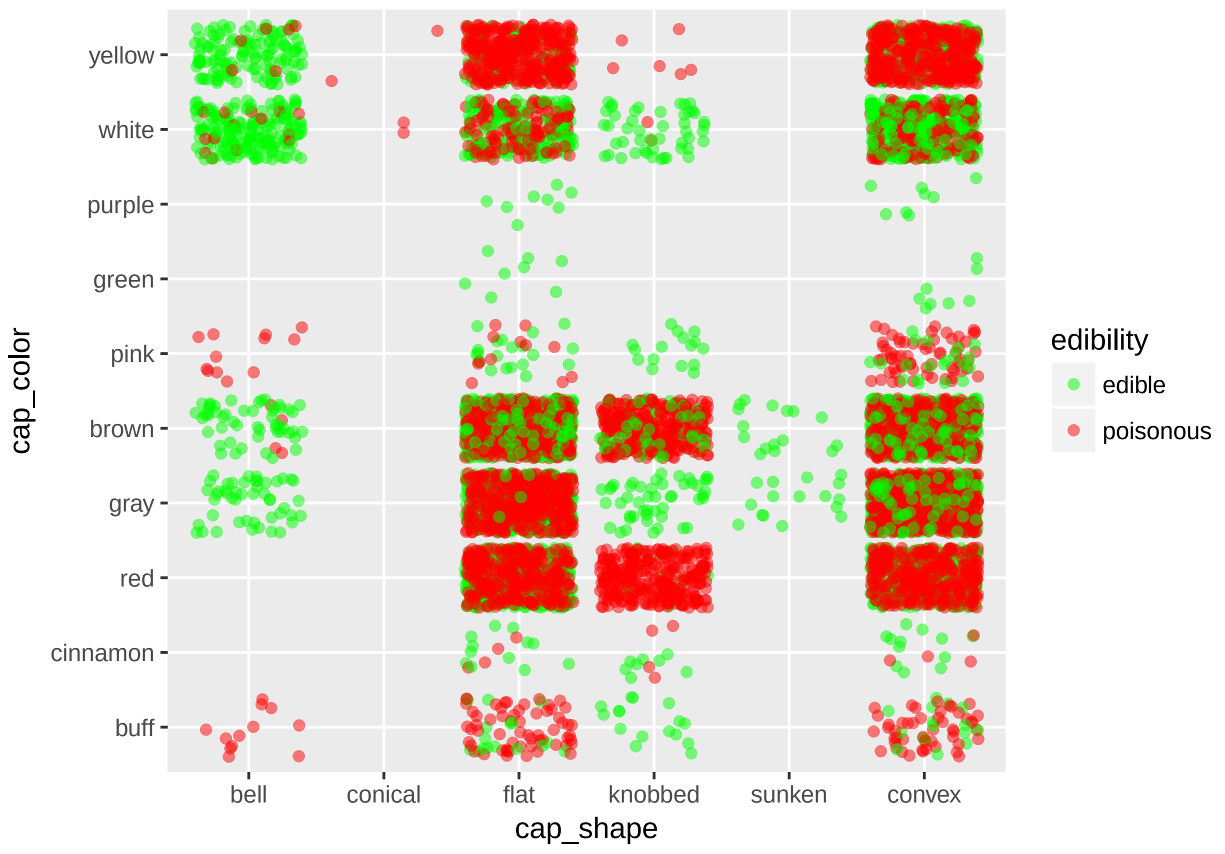
This is one of the most important step in the DS process. This stage can gives us unexpected insights and often allows us to ask the right questions.

```
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"))
```



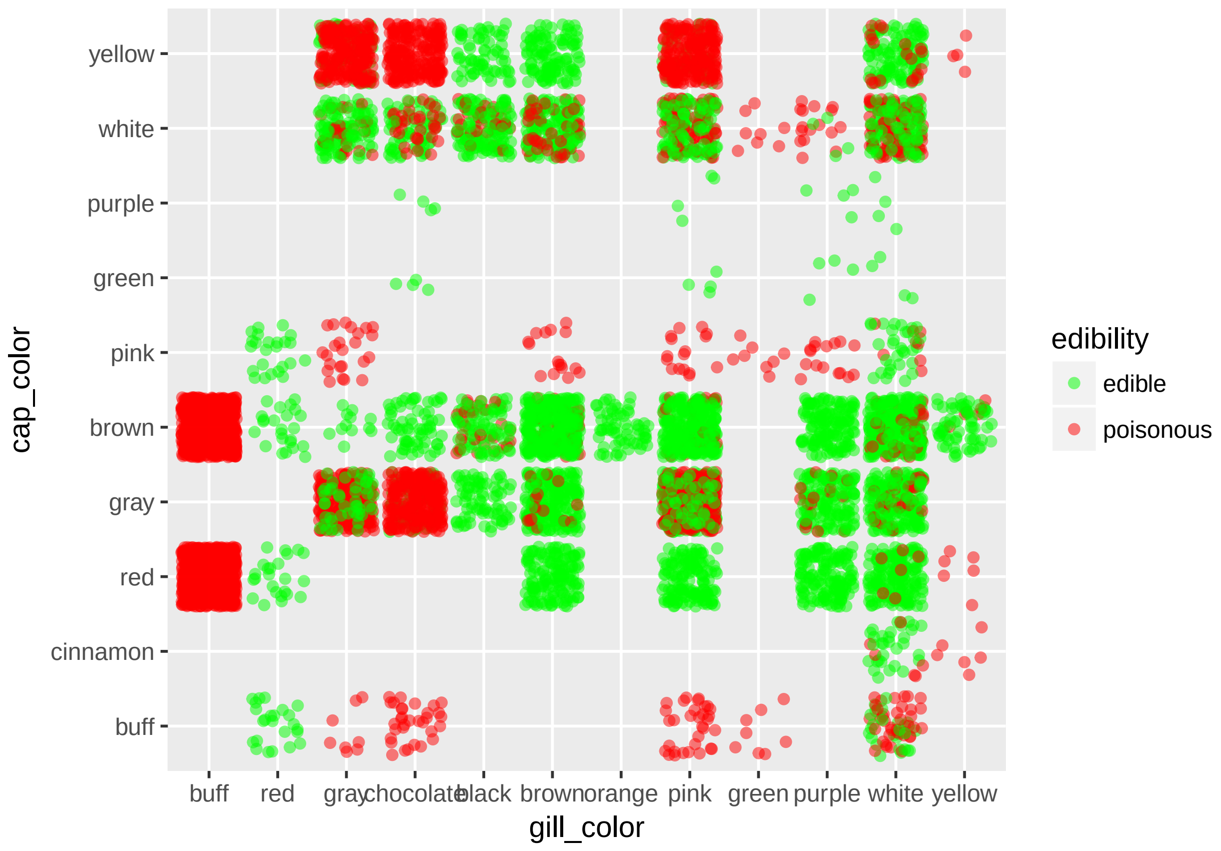
If we want to stay safe, better bet on *fibrous* surface. Stay especially away from *smooth* surface, except if they are purple or green.

```
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"))
```

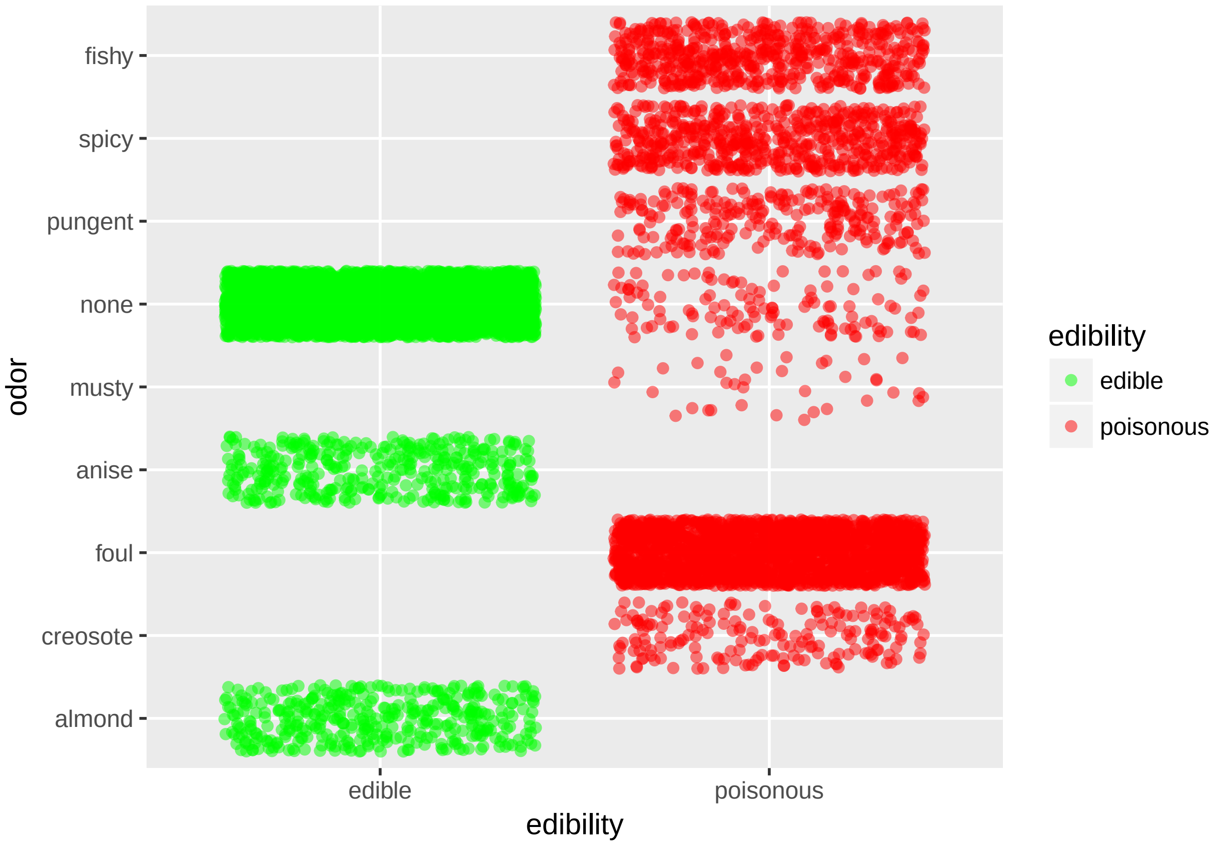


Again, in case one don't know about mushroom, it is better to stay away from all shapes except maybe for *bell* shape mushrooms.

```
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

7.3.3 A. 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, list = FALSE)
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
```

```
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.

7.3.3.1 A. 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 edible (0.98796
##     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 1.00000000) *
```

```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.

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

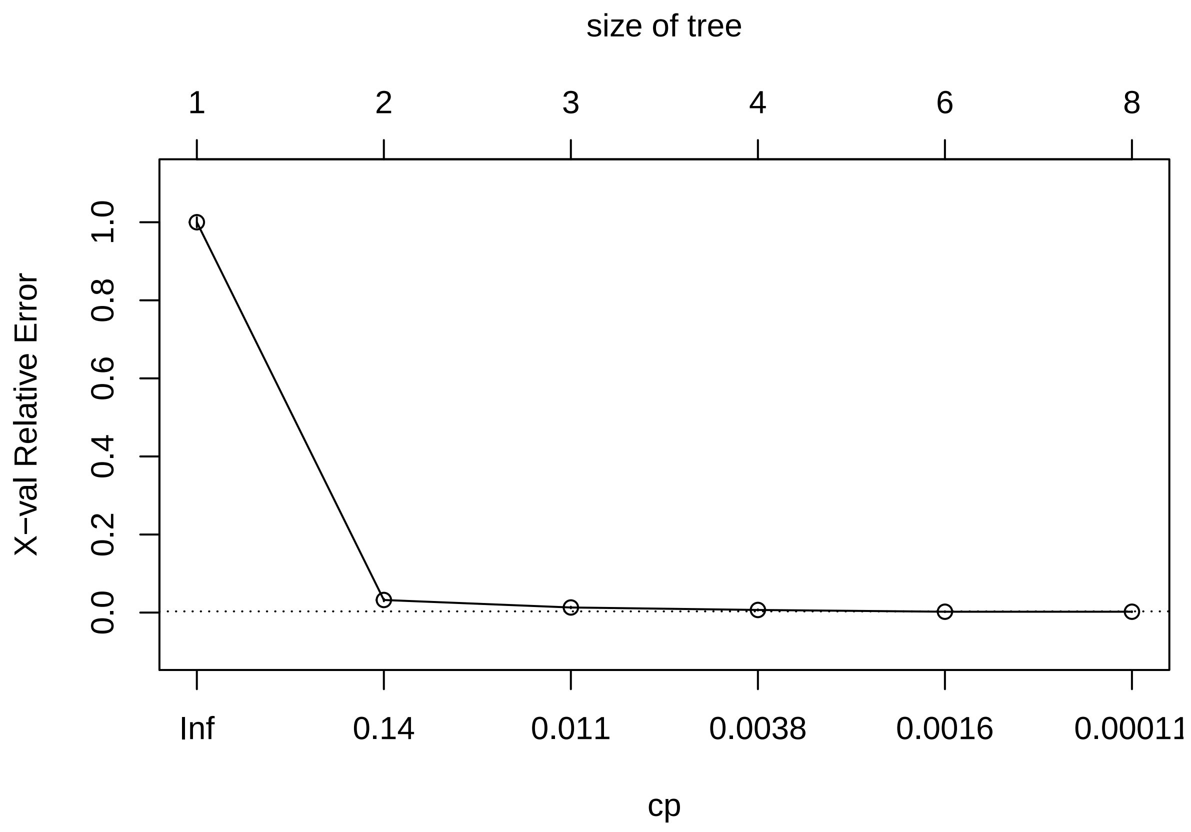
To prune a tree, we first have to find the `cp` that gives the lowest `xerror` or cross-validation error. We can find the lowest `xerror` using either the `printcp` or `plotcp` function.

```
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.

```
plotcp(model_tree)
```



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

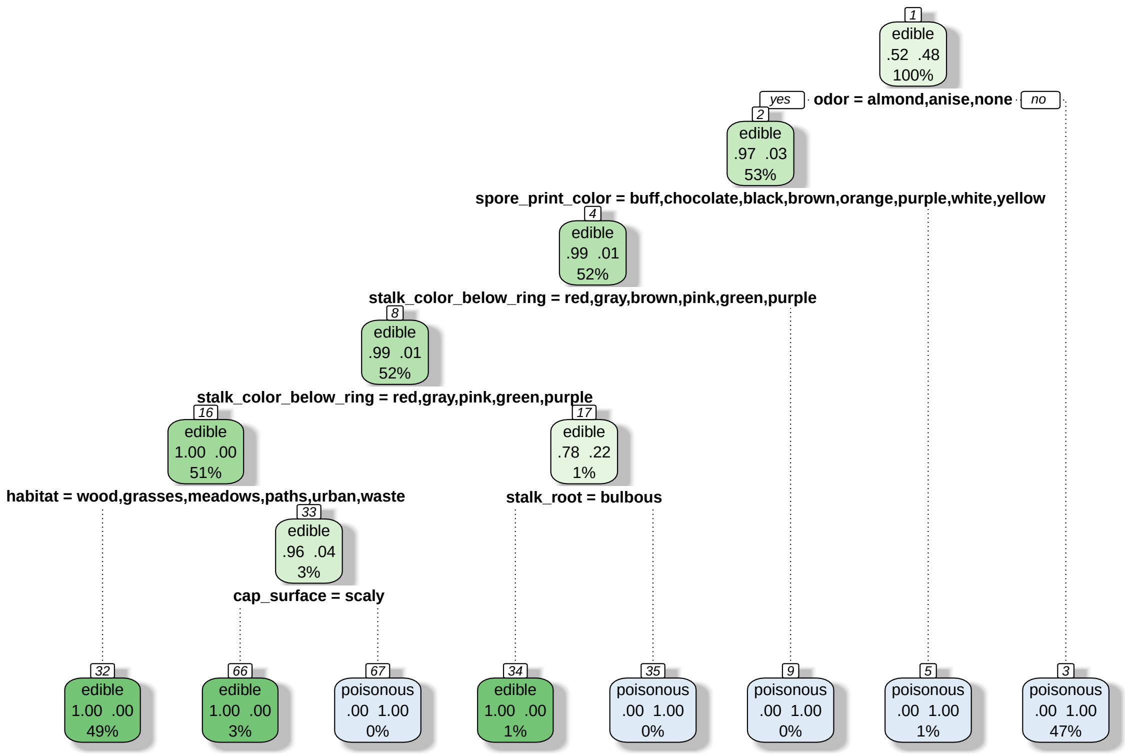
```
## [1] 0.00111714
```

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

```
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

```
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?

```r
#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 = "class"),
                       reference = test_mushroom$edibility,
                       positive = "edible")
```

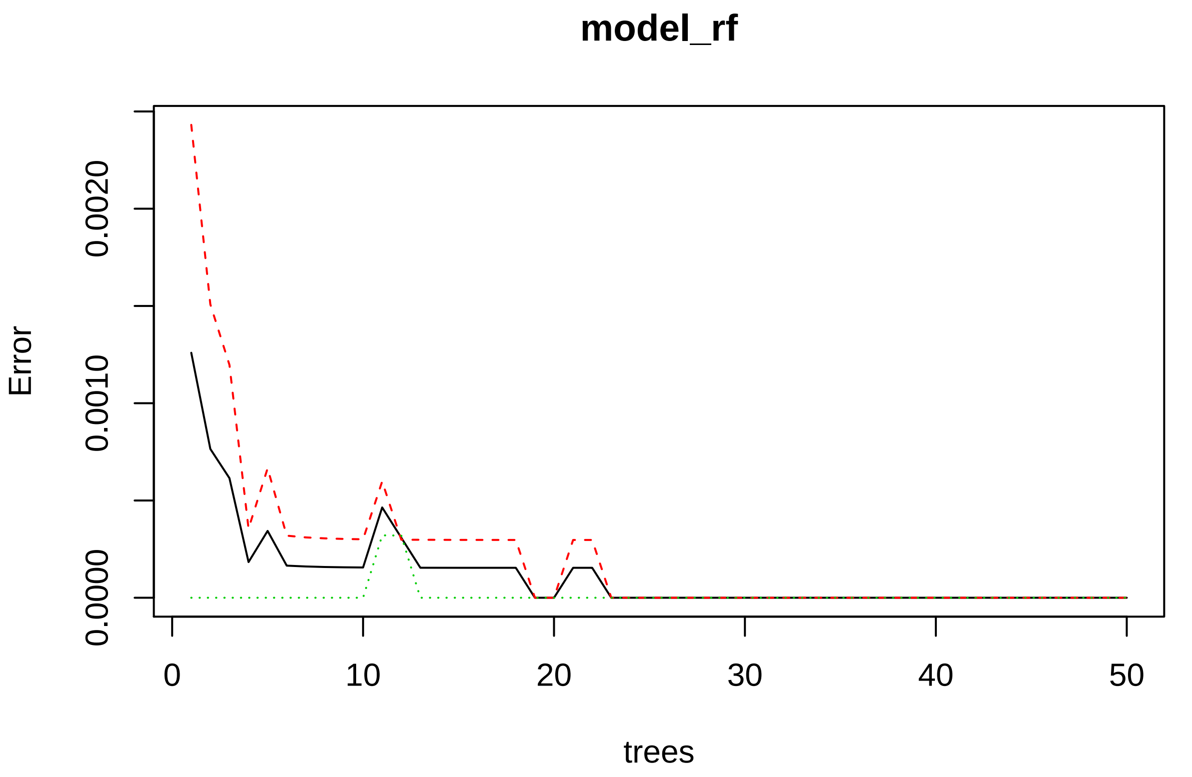
```
## Confusion Matrix and Statistics
##
##            Reference
## Prediction  edible poisonous
##   edible       841         0
##   poisonous      0       783
##
##                Accuracy : 1
##                  95% CI : (0.9977, 1)
##     No Information Rate : 0.5179
##     P-Value [Acc > NIR] : < 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.

7.3.3.2 A. 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.

```
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.

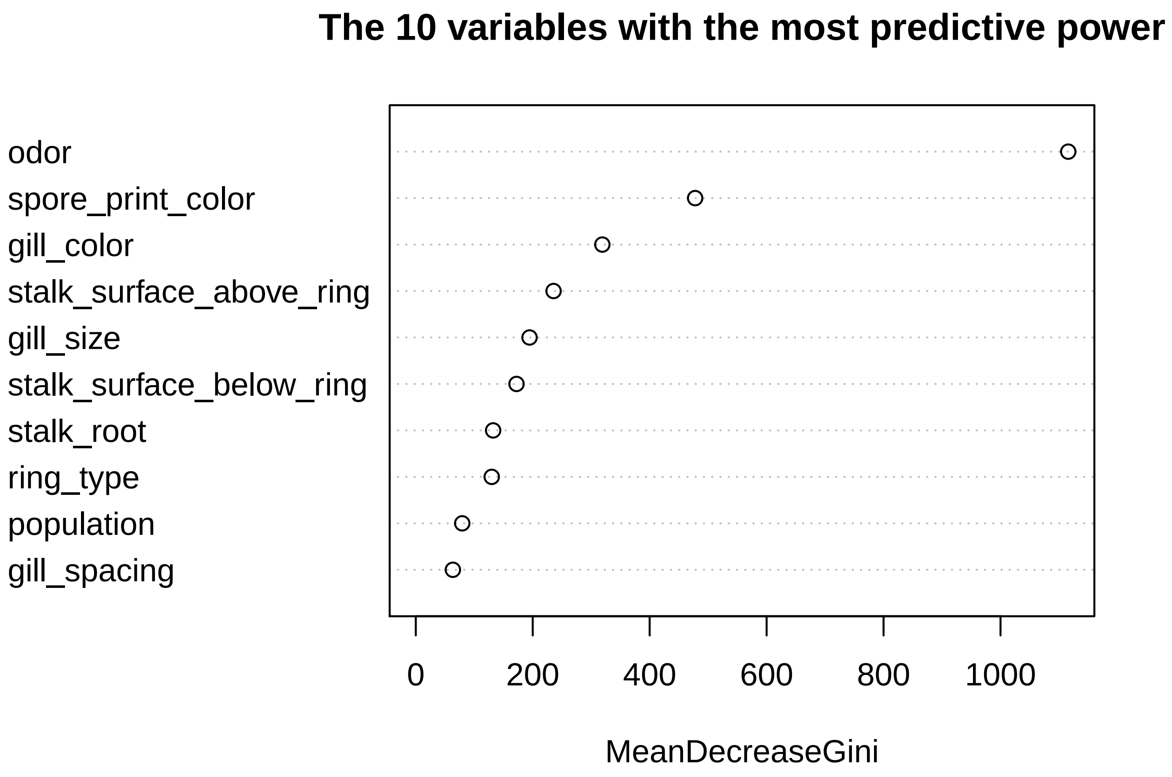
```r
caret::confusionMatrix(data = model_rf$predicted, reference = train_mushroom$edibility ,
                       positive = "edible")
```

```
## 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*

```r
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.

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

```
## # A tibble: 10 × 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!

7.3.3.3 A. Use of SVM

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

Check the prediction

```r
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!

7.4 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.

We would love to hear from you. Give us feedback below.

7.5 Example one

7.6 Example two

Chapter 8

Case Study - Predicting Survicalship 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

8.1 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("~/Google Drive/Software/R projects/datasets/Kaggle_Titanic_train.csv")
test_set <- read_csv("~/Google Drive/Software/R projects/datasets/Kaggle_Titanic_test.csv")

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

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

```
## 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", ...
```

8.2 Tidy the data

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

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

8.3 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.

8.3.1 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.

8.3.1.1 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(mydata$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.
mydata$title <- gsub('(.*,)|(\\..*)', "", mydata$Name)
table(mydata$Sex,mydata$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.

```
mydata$title <- gsub("Mlle", "Miss", mydata$title)
mydata$title <- gsub("Mme", "Mrs", mydata$title)
mydata$title <- gsub("Ms", "Miss", mydata$title)
mydata$title <- gsub("Jonkheer", "Mr", mydata$title)
mydata$title <- gsub("Capt|Col|Major", "Army", mydata$title)
mydata$title <- gsub("Don|Dona|Lady|Sir|the Countess", "Nobility", mydata$title)
```

```
mydata$title <- gsub("Dr|Rev", "Others", mydata$title)
mydata$title <- factor(mydata$title)
mydata$title <- factor(mydata$title,
                  levels(mydata$title)[c(5, 3, 2, 4, 7, 1, 6)] )
table(mydata$Sex, mydata$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.

```
round(prop.table(table(mydata$Survived, mydata$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.

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

## We can now put this as factor and check how many families.
mydata$last_name <- factor(mydata$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.

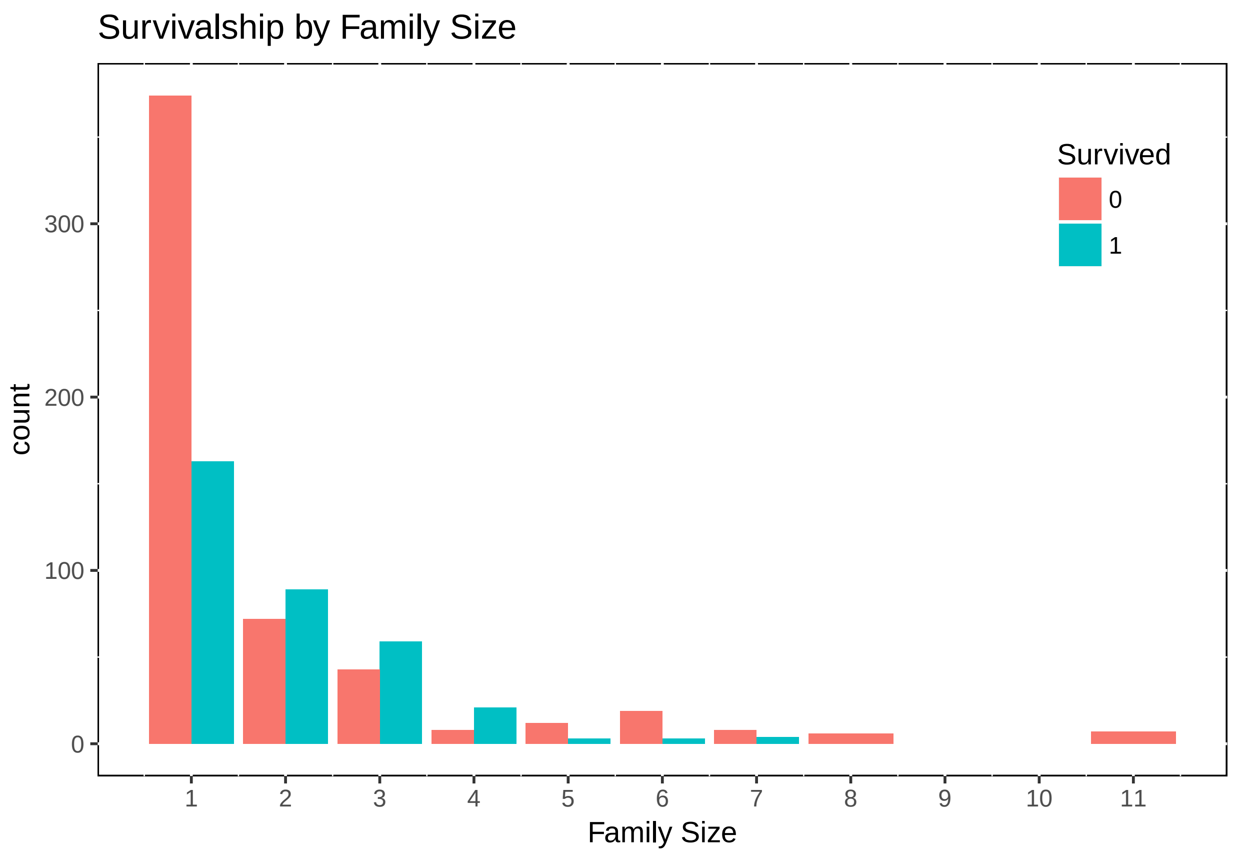
8.3.2 A. Vizualize with families.

We could add a variable about the family size.

```
mydata$family_size <- mydata$SibSp + mydata$Parch + 1
```

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

```
x <- mydata[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.

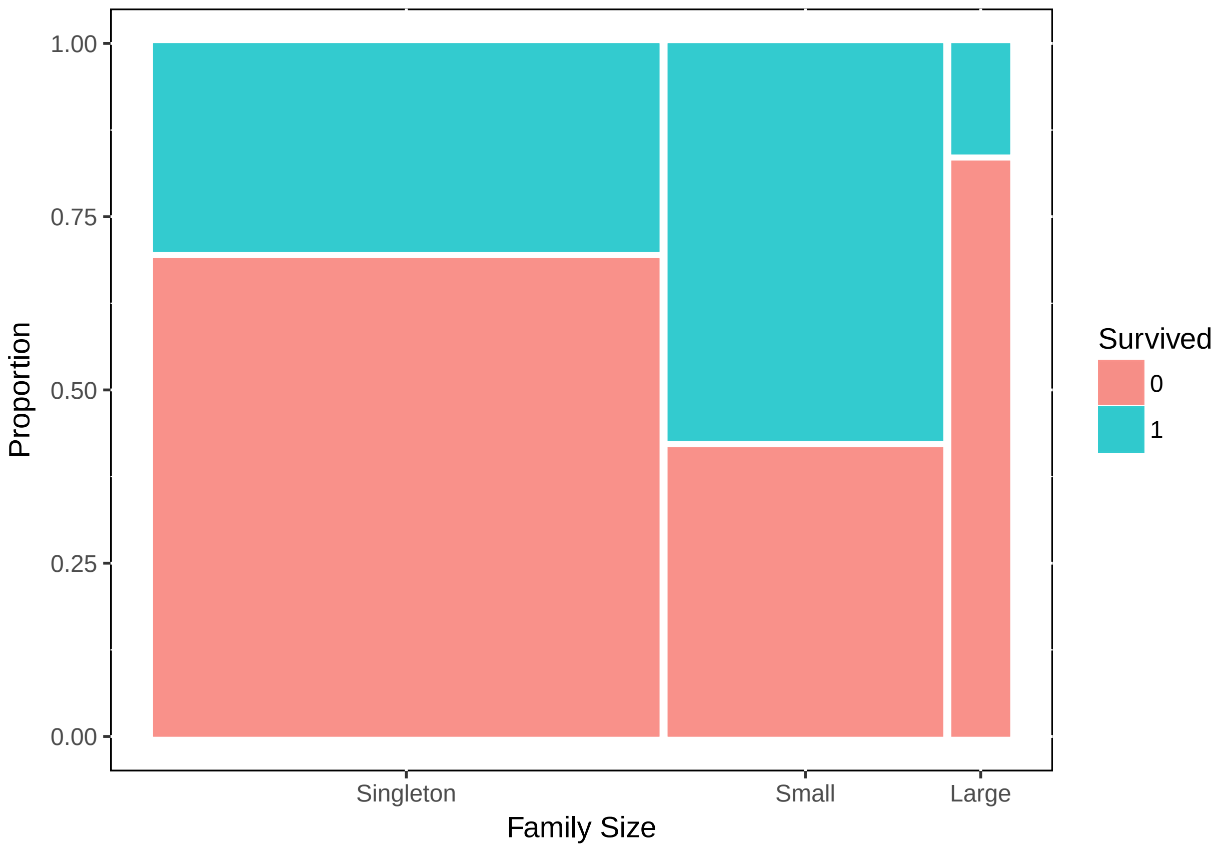
```
mydata$family_size_type[mydata$family_size == 1] <- "Singleton"
mydata$family_size_type[mydata$family_size <= 4 & mydata$family_size > 1] <- "Small"
mydata$family_size_type[mydata$family_size > 4] <- "Large"
mydata$family_size_type <- factor(mydata$family_size_type, levels = c("Singleton", "Small",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

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

```
x <- mydata[1:891,]
table(x$Survived, x$family_size_type)
```

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

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



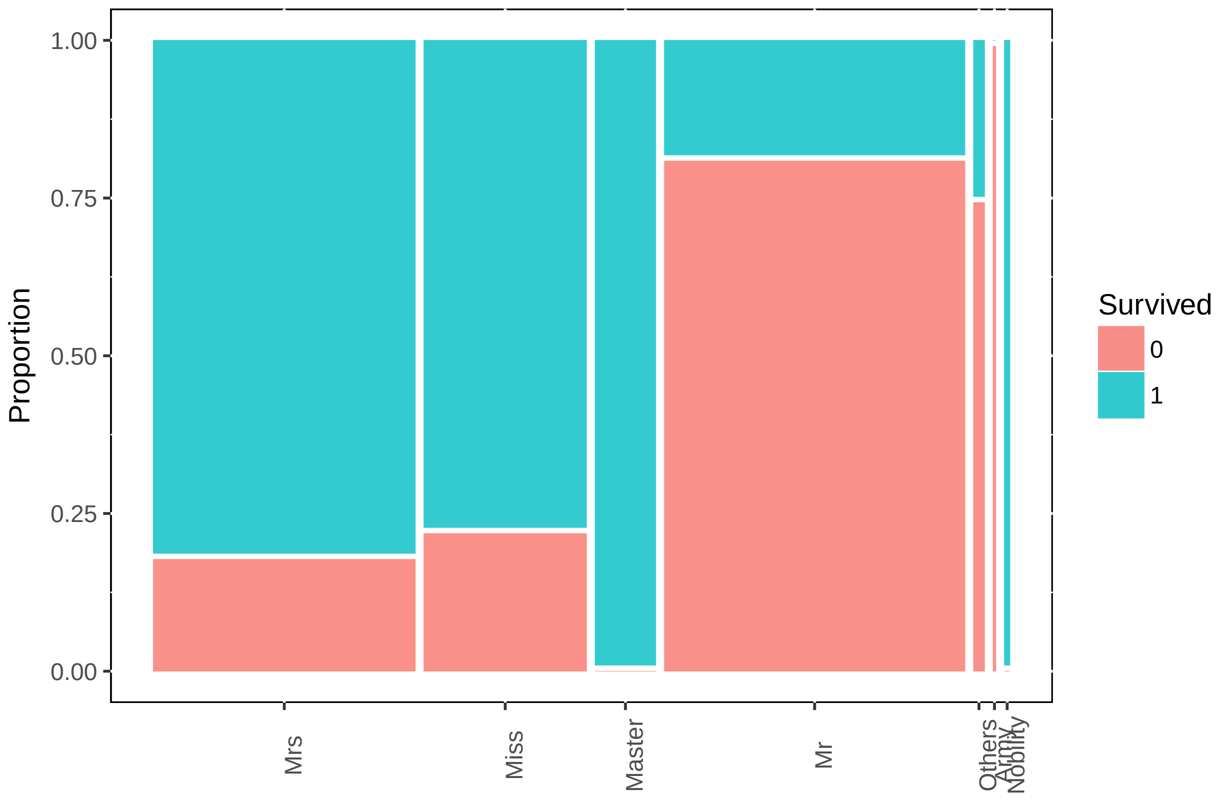
Clearly, there is an advantage in being in a family of size 2, 3 or 4; while there is a disadvantage in being part of of a bigger family.

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?

```
y <- x %>% dplyr::filter(family_size_type == "Small")
table(y$Survived, y$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 = y) + 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

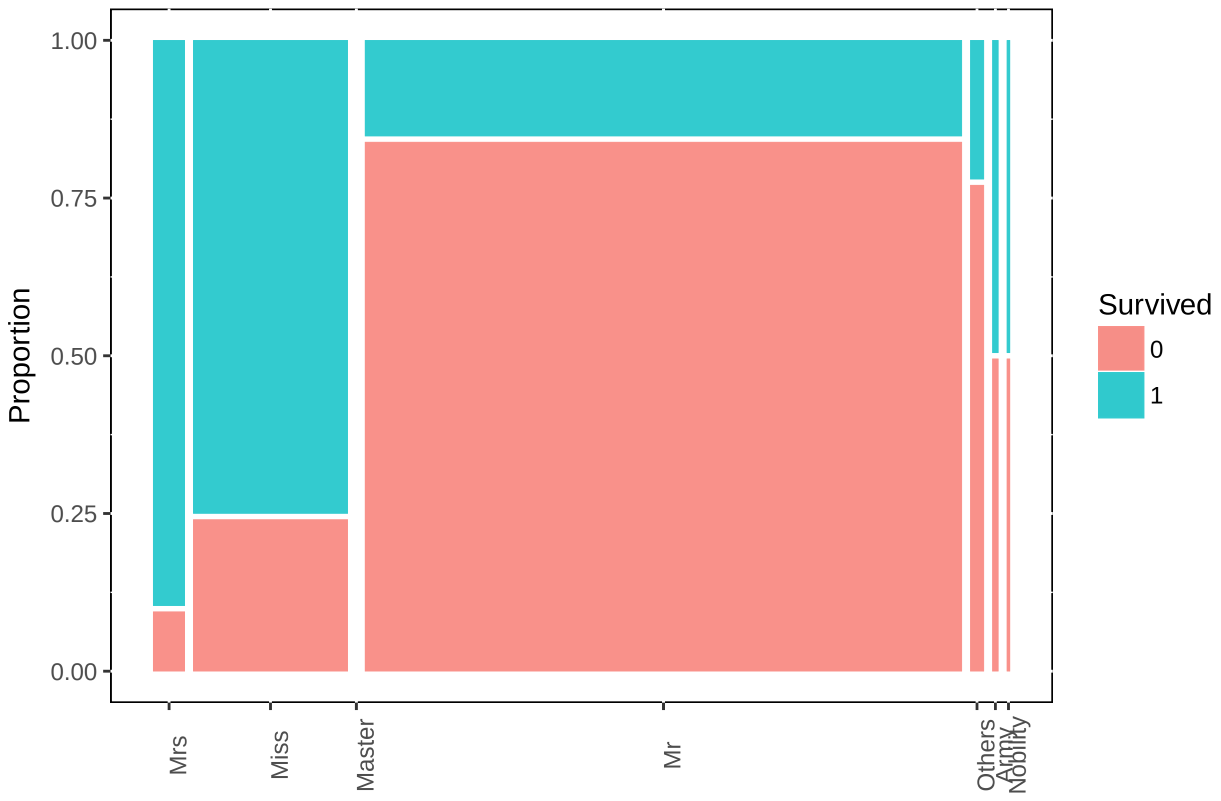
All masters in small families have survived. Miss & Mrs in small family size have also lots of chane of survival.

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

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

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

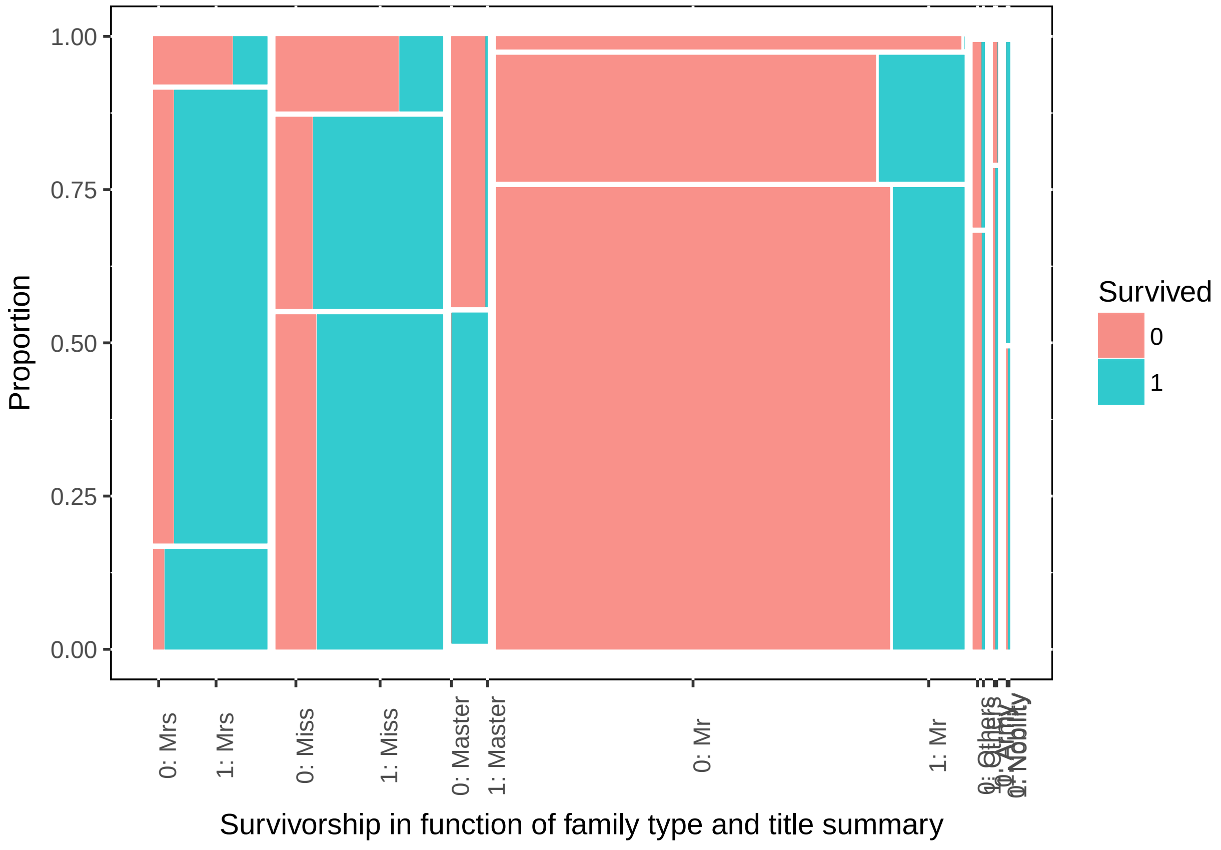
```
ggplot(data = y) + geom_mosaic(aes(x = product(title), fill = Survived)) +
  labs(x = "Survivorship for people who boarded alone in function of their title",
       y = "Proportion") +
  theme(panel.background = NULL, axis.text.x = element_text(angle=90, vjust=1))
```



Survivorship for people who boarded alone in function of their title

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 = x) + geom_mosaic(aes(x = product(family_size_type, title), fill = Survived)) +
  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))
```



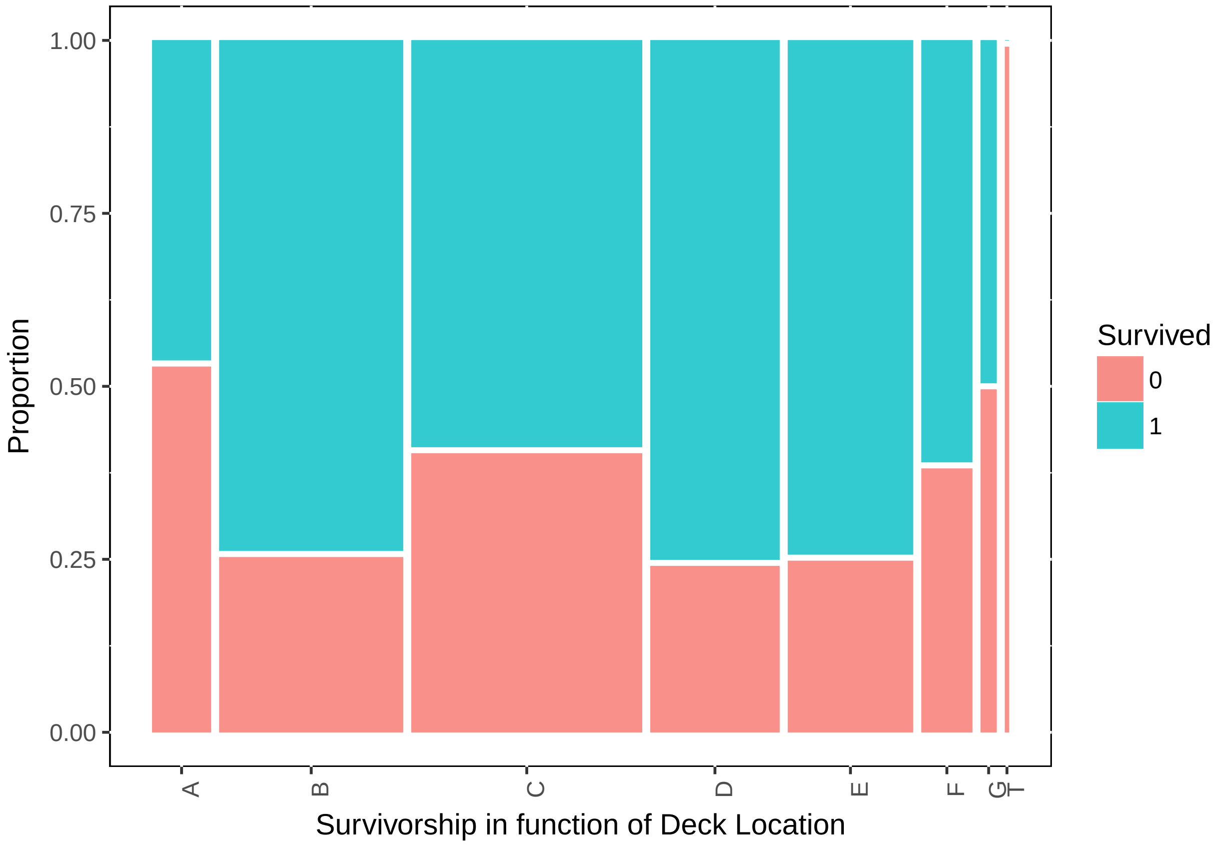
8.4 A. Visualize with cabins.

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.

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

table(x$Survived, x$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 = y) + 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(x$Pclass, x$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(x$Survived, x$Pclass), 2), 2)
```

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

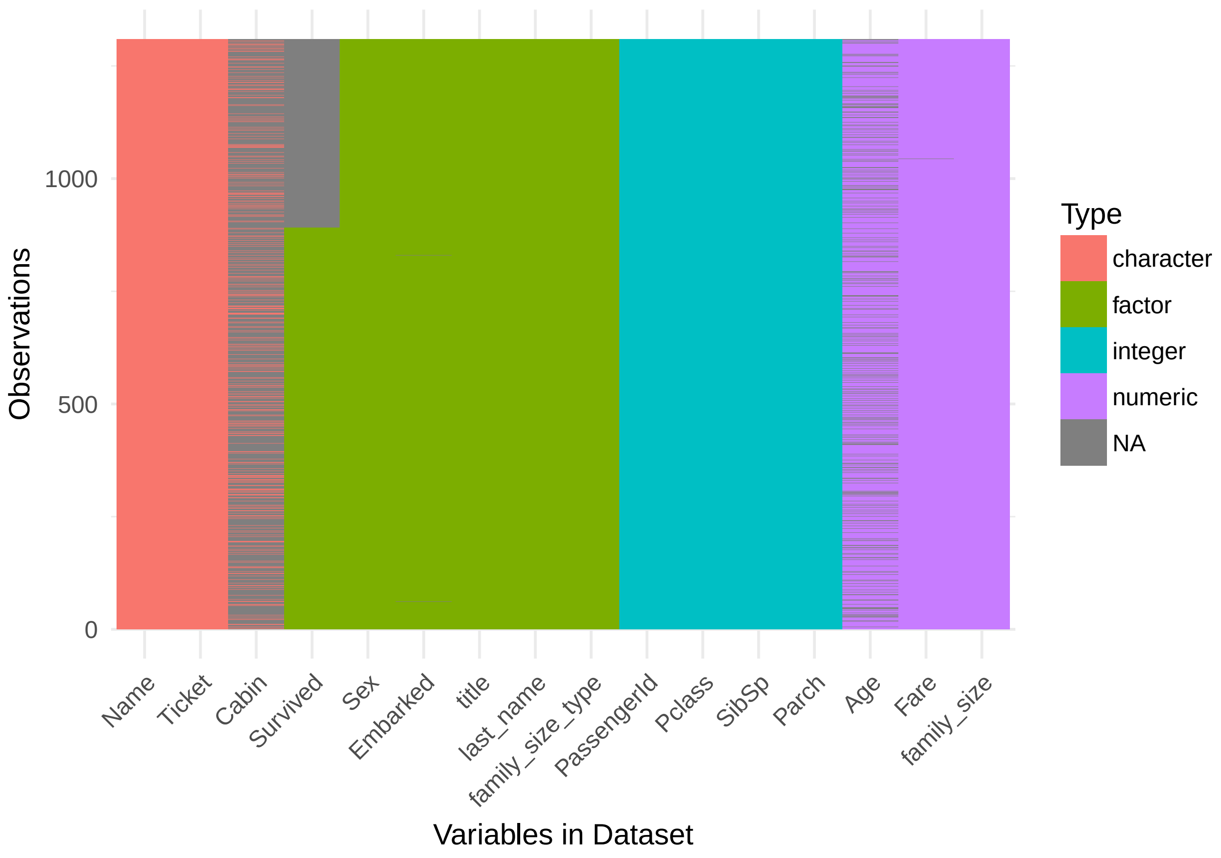
More first class people have survived than other classes.

8.5 B. Transform Dealing with missing data.

8.5.1 Overview.

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

```
visdat::vis_dat(mydata)
```



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(mydata, 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.

8.5.1.1 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.

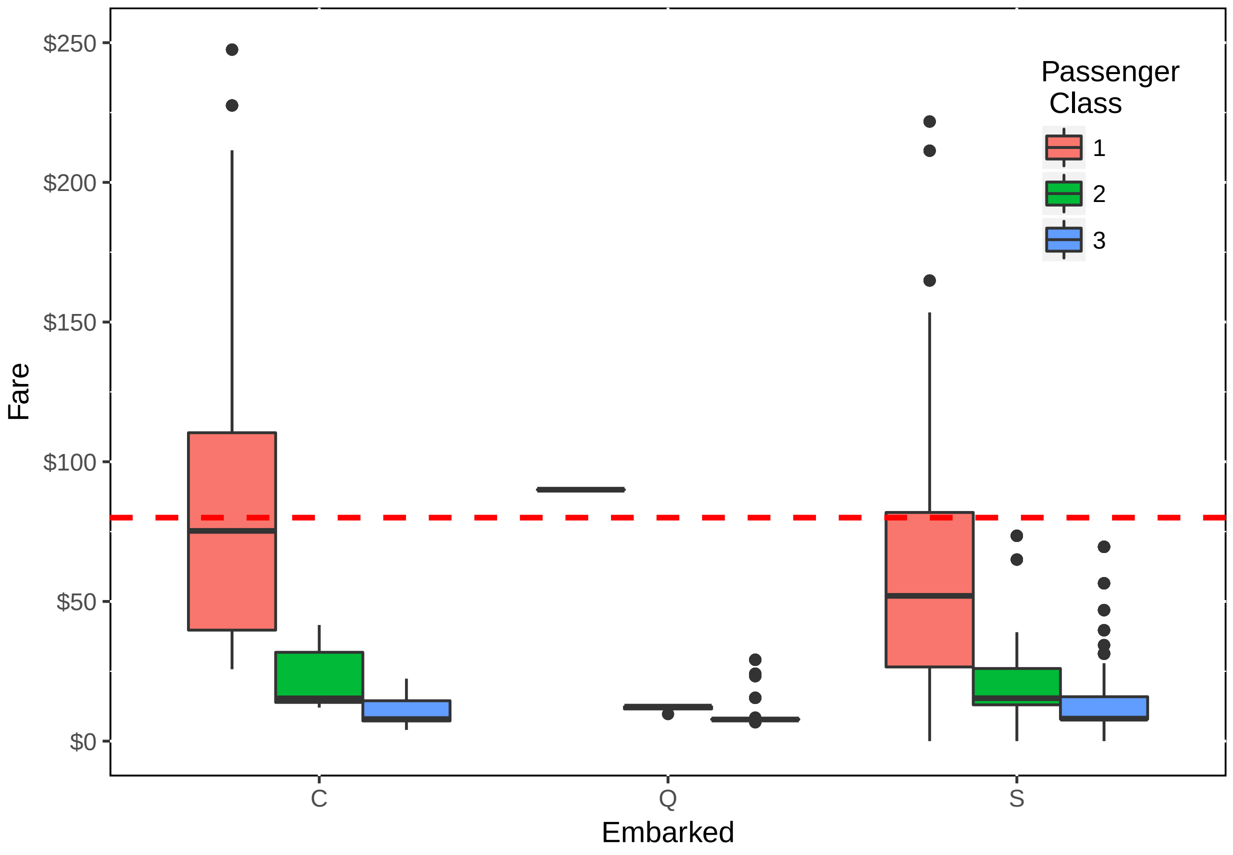
```
y <- which(is.na(mydata$Embarked))
glimpse(mydata[y, ])
```

```
## 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 <- mydata %>% 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.

```
table(mydata$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.

```
x <- mydata %>% 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.

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

Now onto that missing `Fare` data

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

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

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

8.5.1.2 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`

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

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

```
y <- mydata %>% 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)
```

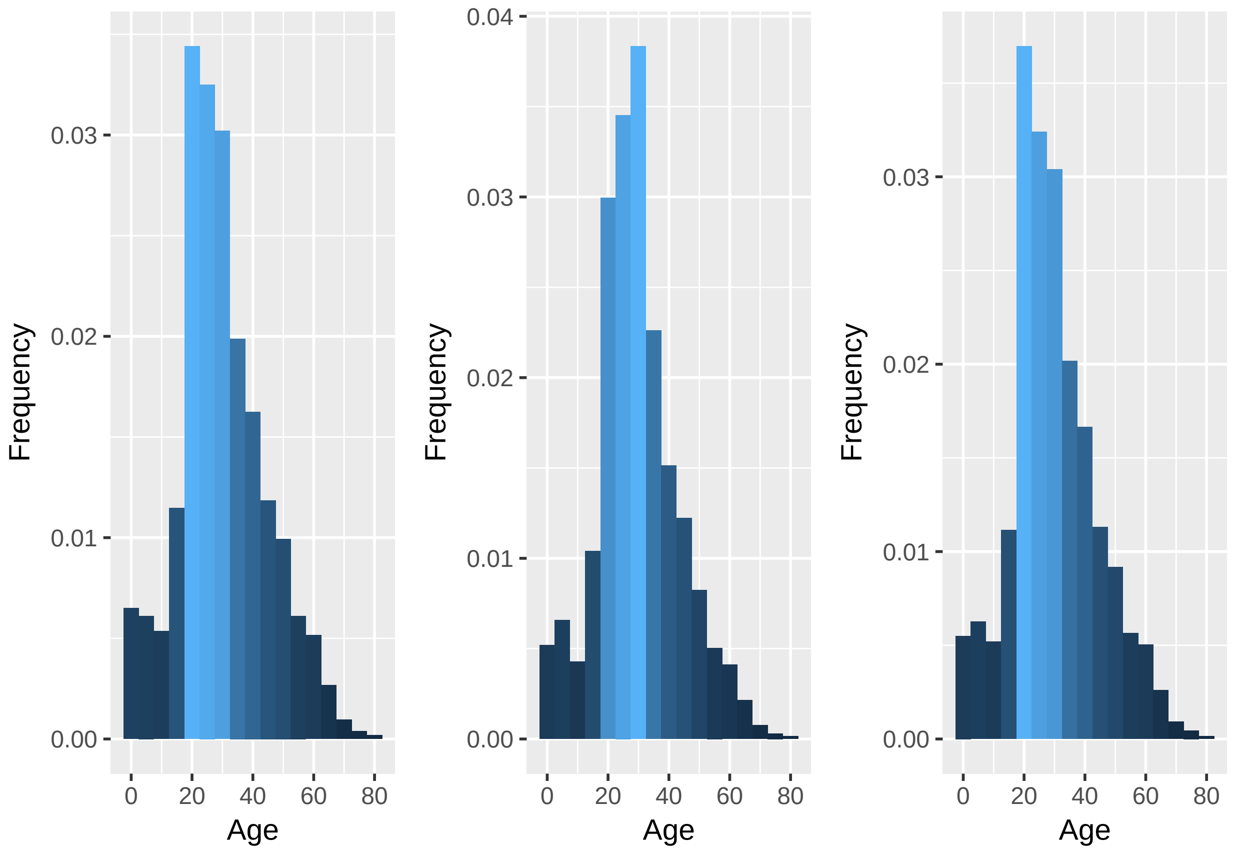
The process is fairly rapid on my computer (around 10~15 seconds)

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

```r
y <- mydata %>% 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.

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

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.

```
mydata$Age <- z2$Age
```

8.5.2 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.

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

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

8.6 References.

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

Chapter 9

Case Study - Text classification: Spam and Ham.

This chapter has been inspired by the Coursera course on Machine Learning Foundations: A Case Study Approach given by Carlos Guestrin and by Emily Fox from Washington University. This course is part of the Machine Learning Specialization

The task was to apply classfification on an Amazon review dataset. Given a review, we create a model that will decide if the review is *positive* (associated with a rating of 4 or 5) or *negative* (associate with a rating of 1 or 2). This is a supervised learning task as the grading associated with the reviews is used as the response variable.

As usual, let's first start by loading our libraries

```
library(tidyverse)
```

Let's have a quick look at our data.

```
product_review <- readr::read_csv("dataset/toyamazonPhilips.csv")

#Let's have a quick look at the reviews
library(pander)
pandoc.table(product_review[2:4,1:3],
             justify = c('left', 'left', 'center'), style = 'grid')
```

```
##
##
## +--------------------------+--------------------------------+----------+
## | name                     | review                         |  rating  |
## +==========================+================================+==========+
## | Philips Avent 3 Pack 9oz | If I had not been given a ton  |    2     |
## | Bottles                  | of Avent bottles, I would have |          |
## |                          | chosen some other system.  The |          |
## |                          | leaking is terrible!!!  You    |          |
## |                          | have to buy the disks          |          |
## |                          | separately, you should get     |          |
## |                          | them for free because they are |          |
## |                          | absolutely essential.  The     |          |
## |                          | only way to mix formula in the |          |
## |                          | bottle or transport liquid is  |          |
## |                          | to use the disks in the ring,  |          |
## |                          | then switch to the nipple when |          |
## |                          | you are ready to feed.  The    |          |
## |                          | only reason I give it a two is |          |
## |                          | because I do like that you can |          |
## |                          | pump directly into the bottle  |          |
## |                          | with the ISIS breast pump.     |          |
## |                          | And, I like the sippy cups.    |          |
## +--------------------------+--------------------------------+----------+
## | Philips Avent 3 Pack 9oz | Leaks! Especially difficult to |    1     |
## | Bottles                  | get a tight seal if you use    |          |
## |                          | one hand (while holding baby). |          |
## |                          | A much better design is the    |          |
## |                          | Breast Flow Learning Curve     |          |
## |                          | First Years bottles. Instead   |          |
## |                          | buy The First Years 3pk.       |          |
## |                          | Breastflow 5oz. Bottles These  |          |
## |                          | worked much better for me.     |          |
## +--------------------------+--------------------------------+----------+
## | Philips Avent 3 Pack 9oz | I have been using the Avent    |    5     |
## | Bottles                  | bottle system for six months   |          |
## |                          | and have been completely       |          |
## |                          | satisfied. I introduced an     |          |
## |                          | Avent bottle to my daughter at |          |
## |                          | four weeks old and she         |          |
## |                          | transitioned easily between    |          |
## |                          | breast and bottle. She is      |          |
## |                          | still breastfed in the morning |          |
## |                          | and evenings but receives an   |          |
## |                          | Avent bottle at daycare and    |          |
## |                          | has never had a problem. I     |          |
## |                          | have never had a bottle leak   |          |
## |                          | of which other consumers have  |          |
## |                          | complained. I would recommend  |          |
## |                          | this system to any parent,     |          |
## |                          | especially those of part-time  |          |
## |                          | breastfed babies.              |          |
## +--------------------------+--------------------------------+----------+
```

```r
#Let's see the table of ratings.
table(product_review$rating)
```

```
##
##  1  2  3  4  5
## 45 33 17 30 66
```

Interestingly the ratings on the Avent Bottles are quite spread on the extreme. It might be that people only write reviews if they are super excited or very frustrated with a product. Because we want this to be a binary classification exercise, we'll do some transformation on these ratings.

```r
# We'll put a 1 for great reviews (4 or 5) or a 0 for bad reviews (1 or 2)
# We remove all the reviews that have a rating of 3
product_review <- product_review %>% filter(rating != 3) %>%
                    mutate(rating_new = if_else(rating >= 4, 1, 0))
product_review_training <-  product_review[1:150, ]
```

Now we create our corpus, then tokenize it, then make it back to a data frame.

```r
library(tm)
corpus_toy <- Corpus(VectorSource(product_review_training$review))
tdm_toy <- DocumentTermMatrix(corpus_toy, list(removePunctuation = TRUE,
                                               removeNumbers = TRUE))

training_set_toy <- as.matrix(tdm_toy)

training_set_toy <- cbind(training_set_toy, product_review_training$rating_new)

colnames(training_set_toy)[ncol(training_set_toy)] <- "y"

training_set_toy <- as.data.frame(training_set_toy)
training_set_toy$y <- as.factor(training_set_toy$y)
```

Now that we have our data frame ready, let's create our model using the `svmLinear3` method.

```r
review_toy_model <- caret::train(y ~., data = training_set_toy, method = 'svmLinear3')
```

Now we try our model on new review data

```r
test_review_data <- product_review[151:174, ]

test_corpus <- Corpus(VectorSource(test_review_data$review))
test_tdm <- DocumentTermMatrix(test_corpus, control=list(dictionary = Terms(tdm_toy)))
test_tdm <- as.matrix(test_tdm)

#Build the prediction
model_toy_result <- predict(review_toy_model, newdata = test_tdm)

check_accuracy <- as.data.frame(cbind(prediction = model_toy_result,
                                      rating = test_review_data$rating_new))

check_accuracy <- check_accuracy %>% mutate(prediction = as.integer(prediction) - 1)

check_accuracy$accuracy <- if_else(check_accuracy$prediction == check_accuracy$rating, 1, 0)
round(prop.table(table(check_accuracy$accuracy)), 3)
```

```
##
##     0     1
## 0.458 0.542
```

Another way to deal with text classification is to use the `RtextTool` library.
We can use the same dataframe that we used in our previous method. Like before we "DocumentTermMatrix", we create a matrix of terms

```r
library(RTextTools)
```

```
## Loading required package: SparseM
```

```
##
## Attaching package: 'SparseM'
```

```
## The following object is masked from 'package:base':
##
##     backsolve
```

```r
product_review_matrix <- create_matrix(product_review[,2], language = "English",
                                       removeNumbers = TRUE,
                                       removePunctuation = TRUE,
                                       removeStopwords = FALSE, stemWords = FALSE)

product_review_container <- create_container(product_review_matrix,
                                             product_review$rating_new,
                                             trainSize = 1:150, testSize = 151:174,
                                             virgin = FALSE)

product_review_model <- train_model(product_review_container, algorithm = "SVM")

product_review_model_result <- classify_model(product_review_container, product_review_model)
x <- as.data.frame(cbind(product_review$rating_new[151:174], product_review_model_result$SVM_LABEL))
colnames(x) <- c("actual_ratings", "predicted_ratings")
x <- x %>% mutate(predicted_ratings = predicted_ratings - 1)
round(prop.table(table(x$actual_ratings == x$predicted_ratings)), 3)
```

```
##
## FALSE  TRUE
##  0.25  0.75
```

Chapter 10

Final Words

We have finished a nice book.

Bibliography

Xie, Y. (2015). *Dynamic Documents with R and knitr.* Chapman and Hall/CRC, Boca Raton, Florida, 2nd edition. ISBN 978-1498716963.

Xie, Y. (2016). *bookdown: Authoring Books and Technical Documents with R Markdown.* R package version 0.3.9.