Data Analytics Applications

Ashwin Malshe

2019-06-25

Contents

Introduction

This book serves as a companion for my Data Analytics Applications course. The objective of the course is to teach students how to use their statistical and machine learning models to analyze real world data and generate actionable insights. Although not all the exercises in this book will have actionable insights for managers, my focus is on business applications more than anything else. This is because I am a business management professor.[1]

I am in the process of making the code files available to everyone. I will update this text once I do so.

In this chapter, you will find information on various miscellaneous topics. Please go over the table content on the left sidebar to familiarize with the topics we will cover.

0.1 Prerequisites

To run exercises in this book, you will need R installed on your computer. The installation files can be obtained from the Comprehensive R Archive Network (CRAN): https://cran.r-project.org.

Although not required, I strongly recommend installing RStudio IDE in order to run R: https://www.rstudio.com/products/rstudio/download/

The book uses several data sets, which are available from various websites. If you plan to use the same data sets, you will need Internet connection to download them.

All the data sets used in this book are also available to download from my Github repository: https://github.com/ashgreat/datasets

The book uses `caret` package for most of the modeling. It is highly recommended that you read the package documentation at https://topepo.github.io/caret/index.html.

0.2 Twitter credentials

The book uses Twitter data for a few exercises. I use the excellent `rtweet` package for accessing Twitter API. You will need an approved developer Twitter account and a functional Twitter app. Mike Kearly, the author of `rtweet` has a detailed post about how to set up an app and get the secret keys. you can follow it here: https://rtweet.info/articles/auth.html. However, the post is a bit old

[1] My personal website has more information about me: https://www.ashwinmalshe.com

and the screenshots don't look like the current Twitter menu. Although all his instructions are still valid, in this section I am showing you current screenshots.[2]

Step 1. Visit https://developer.twitter.com/ and request an account.

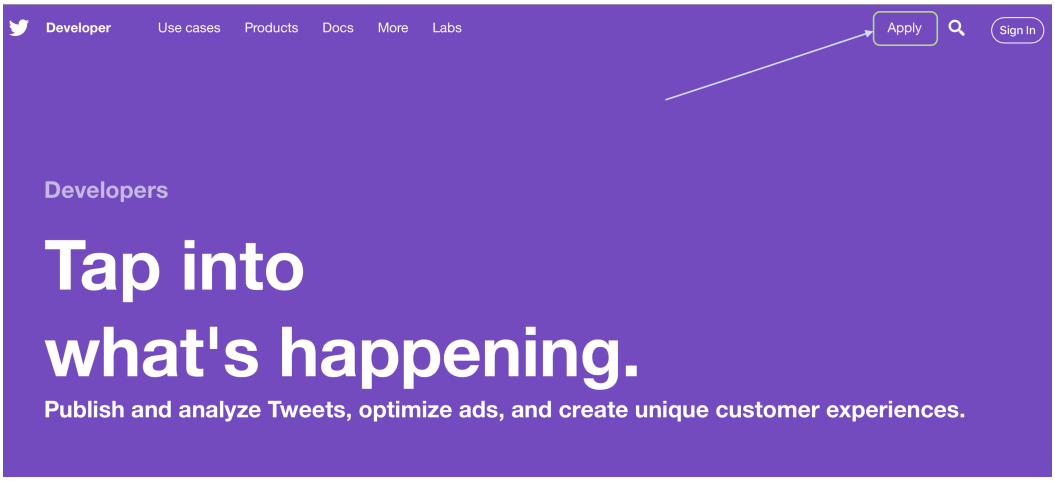


Figure 1: Apply for Developer Account

[2] As of May 15, 2019

Step 2. Once you get the approval, log into the new account and click on your user name on top right to display menu. Click on “Apps”.

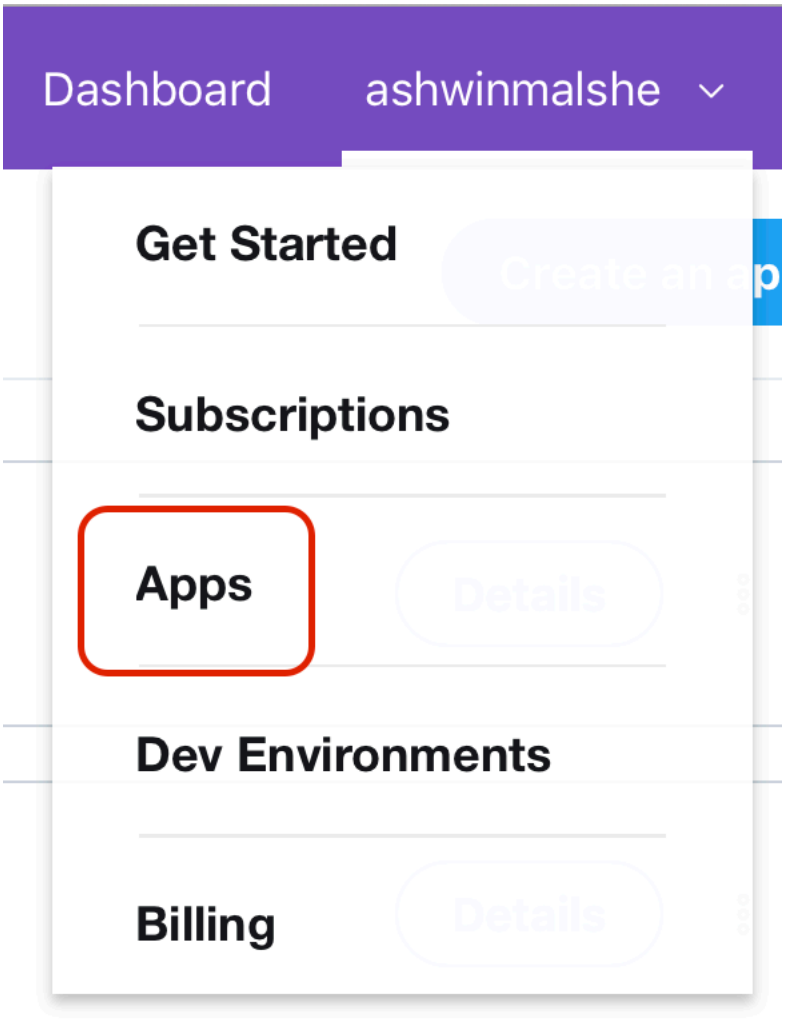


Figure 2: Display the Menu for Apps

Step 3. Click on “Create an app” button.

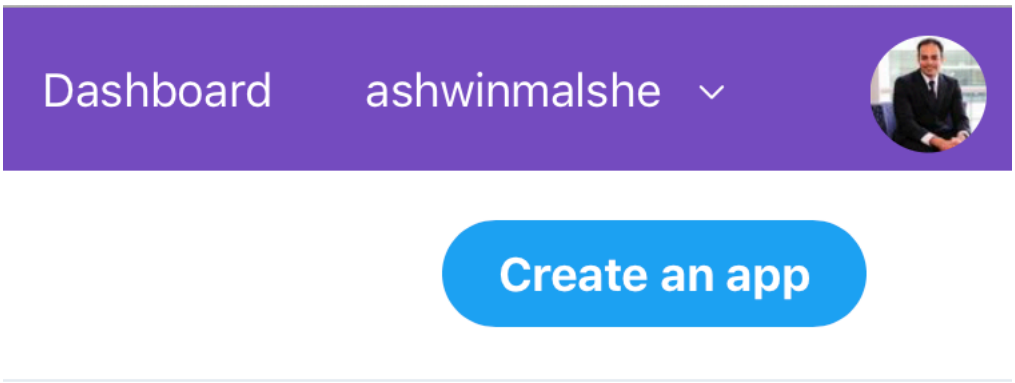


Figure 3: Create an App

Step 4. Fill in your app details. All the required fields need to be filled up.

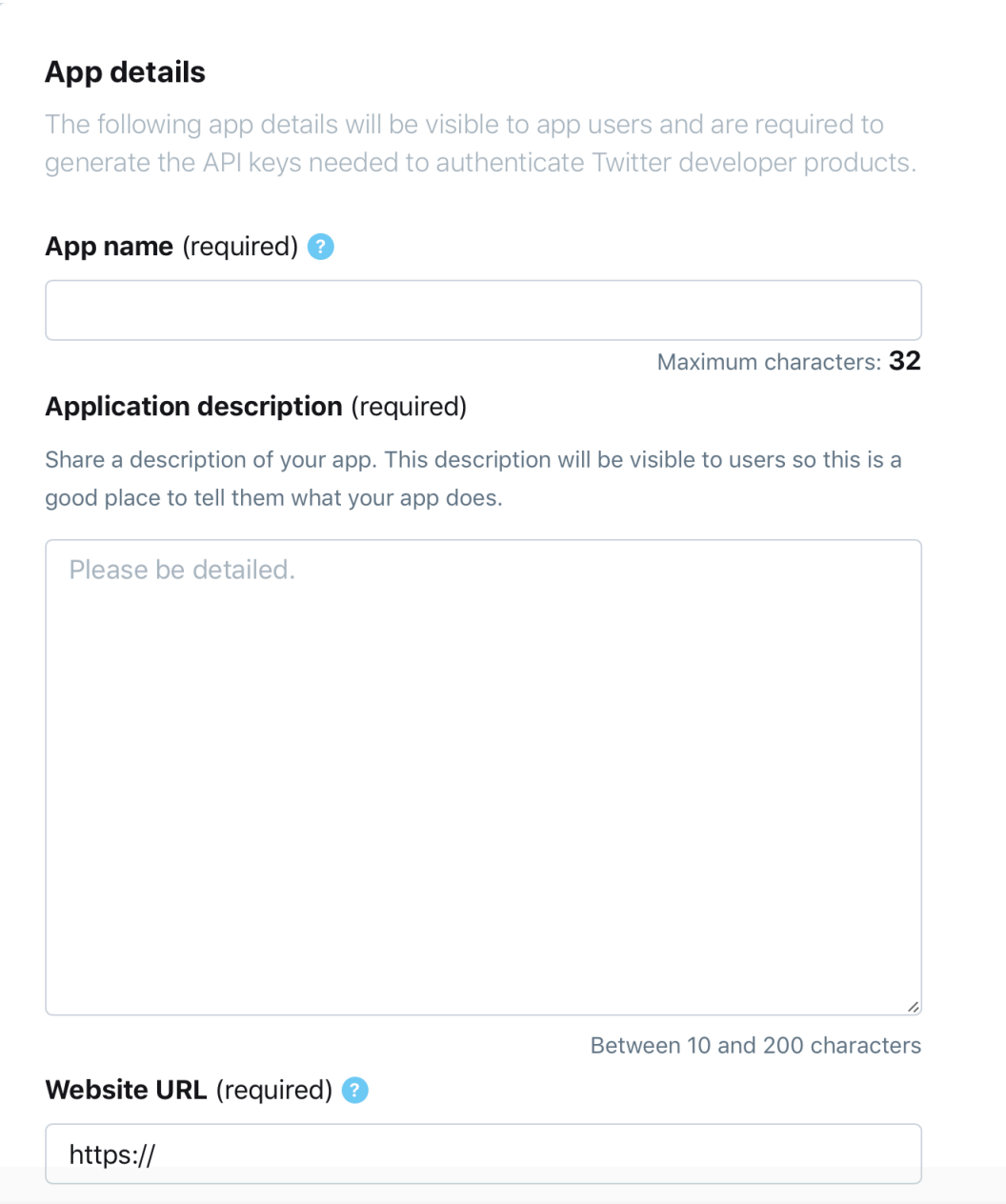


Figure 4: Fill in App Details

Step 5. You will need to provide the following callback URL: http://127.0.0.1:1410.

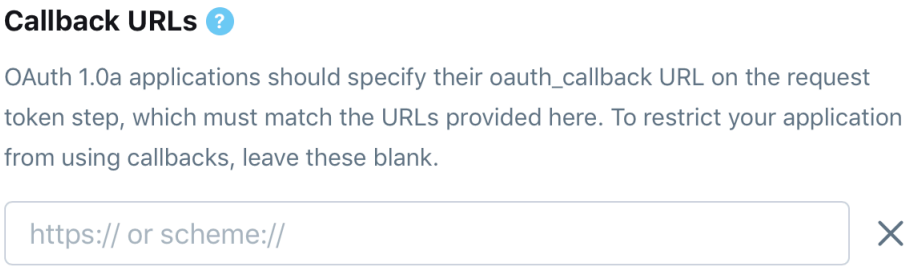


Figure 5: Provide Callback URL

Step 6. Describe how you plan to use this app. "Data collection for academic research and teaching" is a valid use.

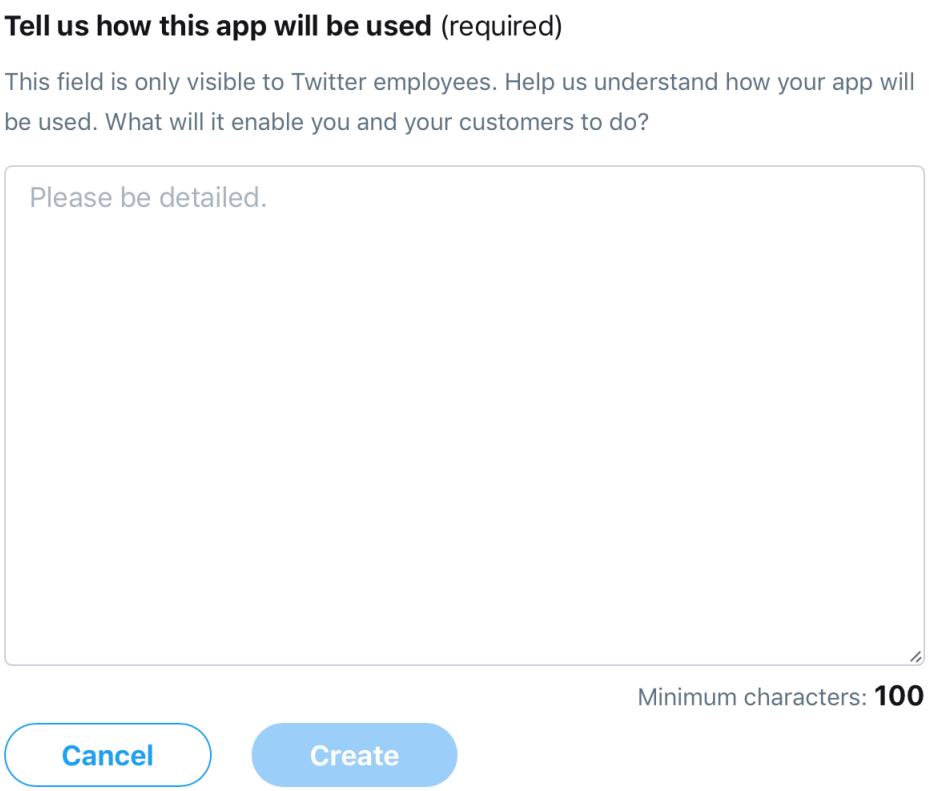


Figure 6: Planned App Use

Step 7. After setting up the app, verify the app details.

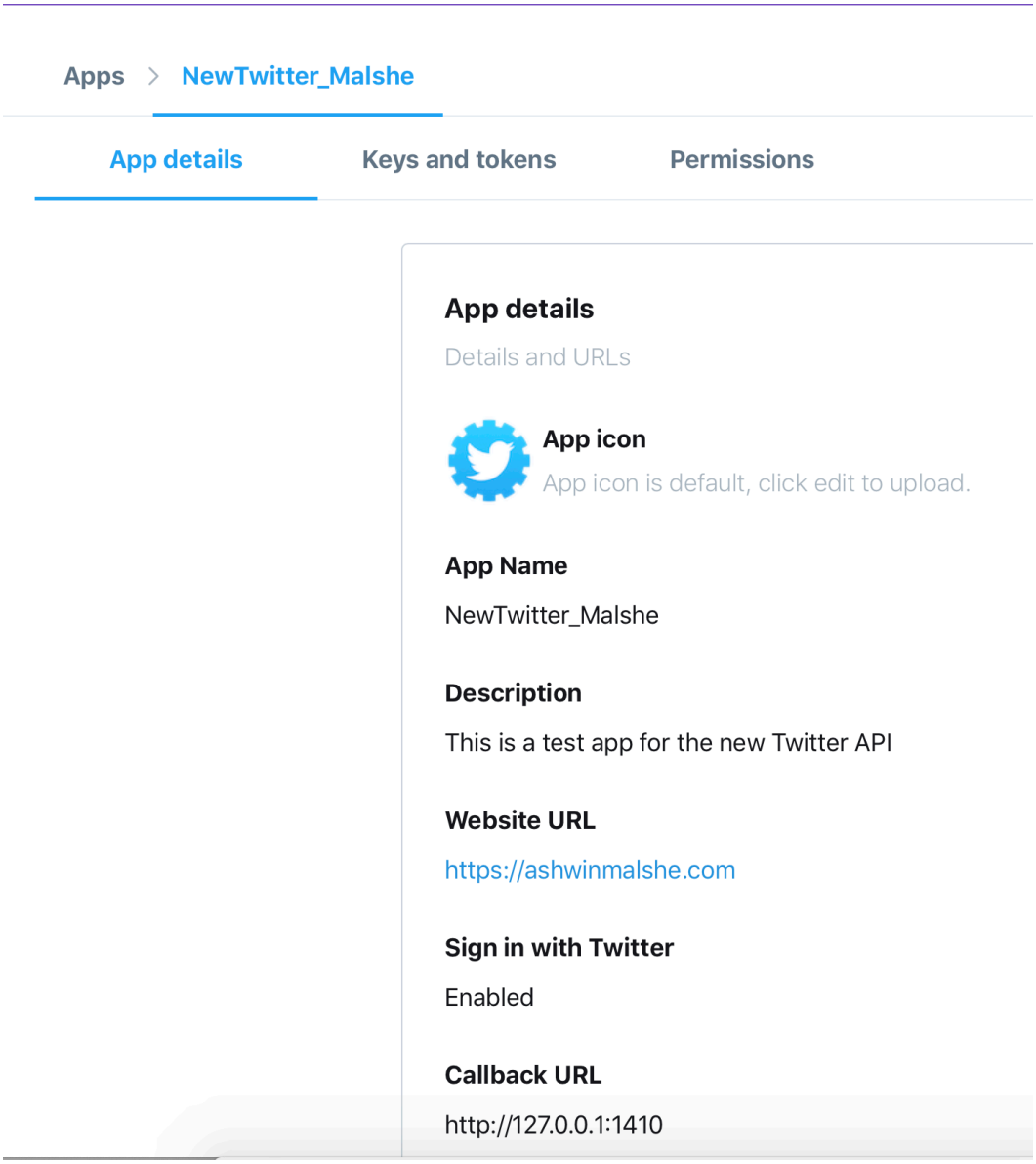


Figure 7: Check App Details

Step 8. Note down the API keys and access tokens. You will need these for accessing the Twitter API.

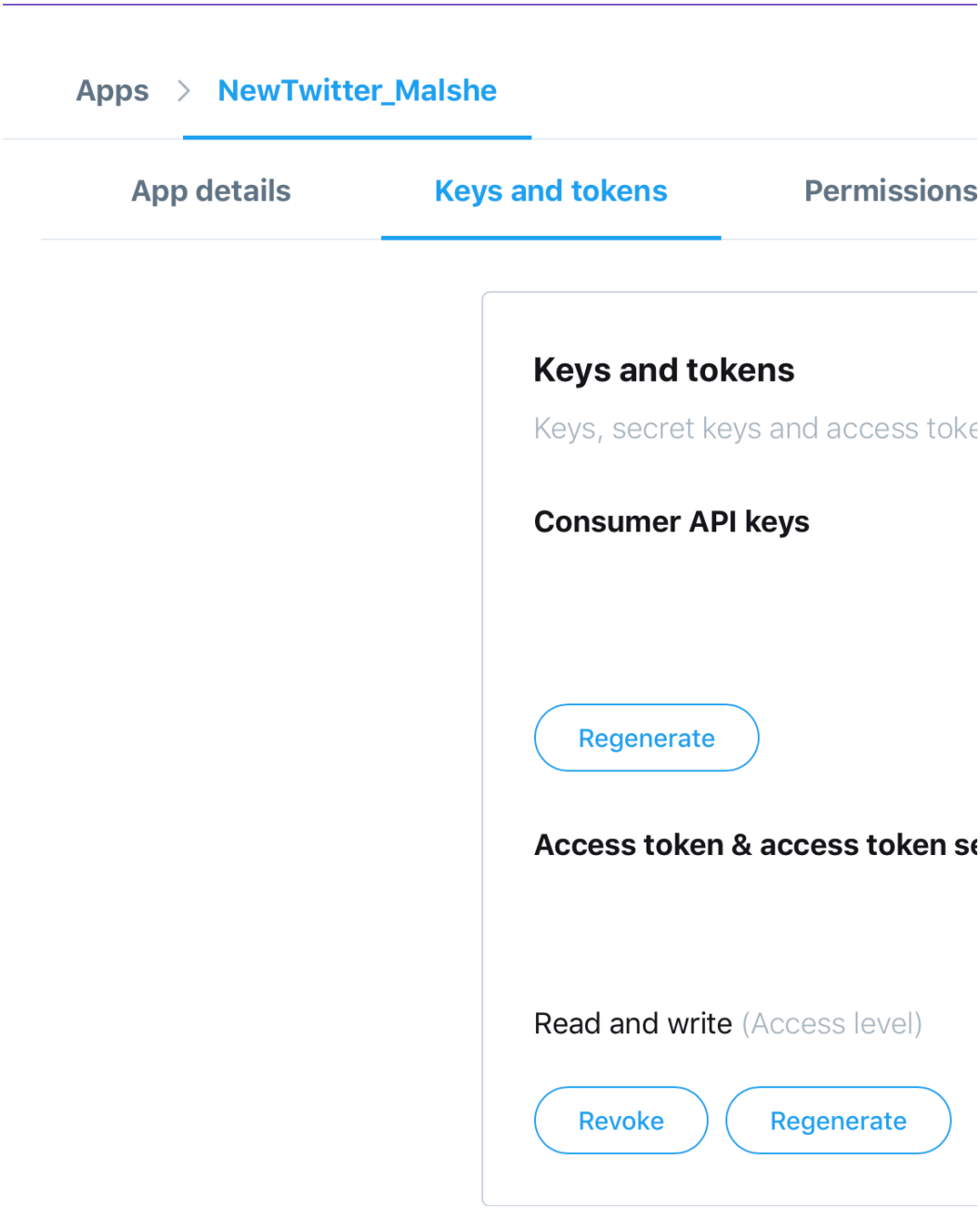


Figure 8: Keys and Tokens

0.3 Model performance metrics

In classification models, the accuracy of predictions is a generally important metric. However, several other metrics may become more important depending on the nature of the application

we deal with.

The following set of model metrics are verbatim reproduced from the help text for `confusionMatrix()` function from `caret` package.

Suppose a 2X2 table with notation

	Reference	
Predicted	Event	No Event
Event	A	B
No Event	C	D

The formulas used here are:

$$Sensitivity = \frac{A}{A+C}$$

$$Specificity = \frac{D}{B+D}$$

$$Prevalence = \frac{A+C}{A+B+C+D}$$

$$PPV = \frac{Sensitivity * Prevalence}{Sensitivity * Prevalence + (1 - Specificity) * (1 - Prevalence)}$$

$$NPV = \frac{Specificity * (1 - Prevalence)}{(1 - Sensitivity) * Prevalence + Specificity * (1 - Prevalence)}$$

$$Detection\ Rate = \frac{A}{A+B+C+D}$$

$$Detection\ Prevalence = \frac{A+B}{A+B+C+D}$$

$$Balanced\ Accuracy = \frac{Sensitivity + Specificity}{2}$$

$$Precision = \frac{A}{A+B}$$

$$Recall = \frac{A}{A+C}$$

$$F1 = \frac{(1+\beta^2) * Precision * Recall}{(\beta^2 * Precision) + Recall}$$

where β = 1 for this function.

Chapter 1

Parameter Tuning

In machine learning, models rely on hyperparameters that we need to tune before finalizing on a model. As this book will extensively use `caret` package, it's important to understand a few key concepts pertaining to hyperparameter tuning that will help you use the package more efficiently.

`caret` enables parameter tuning through two arguments in the `train()` function — `trControl` and `tuneGrid`. We are supposed to pass a function `trainControl()` with its arguments to `trControl` while `tuneGrid` takes a `data.frame` object. Usually, we use `expand.grid()` function from base R to create the grid and pass it on to `tuneGrid`. Let's understand each of these two arguments in more detail.

1.1 Train control

`trainControl()` function takes in several arguments.[1] Here is where we provide `caret` with information on how to train the model. Here is a list of things you can do with `trainControl` –

1. You can specify the method for resampling such as `boot`, `cv`, etc. In most machine learning tasks, we prefer to use cross validation or its variants.[2]
2. You can specify preprocssing options such as scaling continuous values and imputing missing values.
3. You can balance classes using up sampling, down sampling, or SMOTE.[3]

There many other arguments in this function and I strongly recommend you to look at `caret` documentation.

[1] In your RStudio console type `?caret::trainControl()` to get the full syntax and argument description.

[2] Read my note on cross validation if you want to understand its mechanics: http://rpubs.com/malshe/212816

[3] We will see an example of SMOTE in the insurance call data set.

1.2 Grid of hyperparameters

Each machine learning model may have its own set hyperparameters. Some of them have only a few while others may have a lot of parameters to tune. Again, checking out the documentation to understand these hyperparameters is important. Once you know them and the range of values they can take, you can create a grid to do a grid search. A grid is nothing but a full combination of all the possible values of the hyperparameters. In order to create a grid, we use `expand.grid()` function from base R. Here is a simple example.

Consider that we have two parameters `alpha`(α) and `beta`(β) such that $\alpha \in [0, 1]$ and $\beta \in [1, 10]$ As we don't know which combination of these two parameters will give us the best model, we decide to try out multiple combinations and choose one. We decide to increment `alpha` by 0.4 and `beta` by 3 step-wise and try their combinations.

The code below will give us all the possible combinations satisfying the rules we specified.

```
expand.grid(alpha = seq(0, 1, by = 0.4),
            beta = seq(1, 10, by = 3))
```

```
##    alpha beta
## 1    0.0    1
## 2    0.4    1
## 3    0.8    1
## 4    0.0    4
## 5    0.4    4
## 6    0.8    4
## 7    0.0    7
## 8    0.4    7
## 9    0.8    7
## 10   0.0   10
## 11   0.4   10
## 12   0.8   10
```

Thus, we get 12 possible combinations. We can pass this `data.frame` to the `train()` function and it will run models using all the 12 combinations and determine the best performing combination in terms of the metric that we specify.

Chapter 2

Titanic Survival Prediction

In machine learning applications, one of the first exercises is to build a model to classify Titanic survivors. The exercise has a little practical value beyond being a learning exercise. However, there are a lot of interesting findings from this data set.

Kaggle hosted a competition using Titanic data a while back, and it is accessible here: https://www.kaggle.com/c/titanic/data

You can also download CSV files from my Github repository: https://github.com/ashgreat/datasets. We will use these links in the code.

The objectives of this exercise are as follows:

1. Use binary logistic regression model to classify survivors and deaths
2. Use XGBoost to classify survivors and deaths

2.1 Data

Let's start with loading required packages.

```
library(dplyr)
library(ggplot2)
library(caret)
library(mice)
library(psych)
library(doParallel)
```

Read Titanic training and test data files.

```
titanic_train <- read.csv("http://bit.ly/2DVwM0d")
titanic_test <- read.csv("http://bit.ly/2Jn7USt")
```

We will use `titanic_train` for model building and then, if you wish, test the efficacy of the model using `titanic_test`. However, `titanic_test` doesn't have the true values of the dependent variables `survived`. To know whether your classification is good, you will have to submit it on Kaggle and get the score.

Let's find the structure of the data and what it contains.

```
## $strict.width
## [1] "wrap"
##
## $digits.d
## [1] 3
##
## $vec.len
## [1] 4
##
## $drop.deparse.attr
## [1] TRUE
##
## $formatNum
## function (x, ...)
## format(x, trim = TRUE, drop0trailing = TRUE, ...)
## <environment: 0x7fdab733bb98>
```

```
str(titanic_train, vec.len = 2)
```

```
## 'data.frame':    891 obs. of  12 variables:
##  $ PassengerId: int  1 2 3 4 5 ...
##  $ Survived   : int  0 1 1 1 0 ...
##  $ Pclass     : int  3 1 3 1 3 ...
##  $ Name       : Factor w/ 891 levels "Abbing, Mr. Anthony",..: 109 191 358 277 16 ...
##  $ Sex        : Factor w/ 2 levels "female","male": 2 1 1 1 2 ...
##  $ Age        : num  22 38 26 35 35 ...
##  $ SibSp      : int  1 1 0 1 0 ...
##  $ Parch      : int  0 0 0 0 0 ...
##  $ Ticket     : Factor w/ 681 levels "110152","110413",..: 524 597 670 50 473 ...
##  $ Fare       : num  7.25 71.28 ...
##  $ Cabin      : Factor w/ 148 levels "","A10","A14",..: 1 83 1 57 1 ...
##  $ Embarked   : Factor w/ 4 levels "","C","Q","S": 4 2 4 4 4 ...
```

Note that `PassengerId`, `Name`, `Ticket`, and `Cabin` seem like variables that we will not use in modeling. However, as we will see below there is a possibility for using information contained in `Name`. Also notice that `Embarked` has 4 levels but one of them is blank. Take a look at its distribution:

```
table(titanic_train$Embarked)
```

Table 2.1: Variables Description

Variable	Definition	Key
survival	Survival	0 = No, 1 = Yes
pclass	Ticket class	1 = 1st, 2 = 2nd, 3 = 3rd
sex	Sex	
Age	Age in years	
sibsp	# of siblings / spouses aboard the Titanic	
parch	# of parents / children aboard the Titanic	
ticket	Ticket number	
fare	Passenger fare	
cabin	Cabin number	
embarked	Port of Embarkation	C = Cherbourg, Q = Queenstown, S = Southampton

```
##
##       C   Q   S
##   2 168  77 644
```

As only 2 values are missing, we should either drop these observations or we should impute them. An easy fix is to replace them by the mode of the distribution, which is S.

```
titanic_train <- titanic_train %>%
  mutate(Embarked = factor(ifelse(Embarked == "", "S", as.character(Embarked))))
```

The variable description from Kaggle is as shown in Table 2.1

Also, Kaggle provides more information on the variables as follows:

Variable Notes

Pclass: A proxy for socio-economic status (SES) 1st = Upper 2nd = Middle 3rd = Lower

Age: Age is fractional if less than 1. If the age is estimated, is it in the form of xx.5

Sibsp: The dataset defines family relations in this way... Sibling = brother, sister, stepbrother, stepsister Spouse = husband, wife (mistresses and fiancés were ignored)

Parch: The data set defines family relations in this way... Parent = mother, father Child = daughter, son, stepdaughter, stepson Some children traveled only with a nanny, therefore parch = 0 for them.

2.1.1 Missing values

```
psych::describe(titanic_train) %>%
  select(-vars, -trimmed, -mad, -range, -se) %>%
  knitr::kable(digits = 2,
               align = "c",
               caption = "Summary Statistics",
               booktabs = TRUE) # kable prints nice-looking tables.
```

Table 2.2: Summary Statistics

	n	mean	sd	median	min	max	skew	kurtosis
PassengerId	891	446.00	257.35	446.00	1.00	891.00	0.00	-1.20
Survived	891	0.38	0.49	0.00	0.00	1.00	0.48	-1.77
Pclass	891	2.31	0.84	3.00	1.00	3.00	-0.63	-1.28
Name*	891	446.00	257.35	446.00	1.00	891.00	0.00	-1.20
Sex*	891	1.65	0.48	2.00	1.00	2.00	-0.62	-1.62
Age	714	29.70	14.53	28.00	0.42	80.00	0.39	0.16
SibSp	891	0.52	1.10	0.00	0.00	8.00	3.68	17.73
Parch	891	0.38	0.81	0.00	0.00	6.00	2.74	9.69
Ticket*	891	339.52	200.83	338.00	1.00	681.00	0.00	-1.28
Fare	891	32.20	49.69	14.45	0.00	512.33	4.77	33.12
Cabin*	891	18.63	38.14	1.00	1.00	148.00	2.09	3.07
Embarked*	891	2.54	0.79	3.00	1.00	3.00	-1.26	-0.22

Only `Age` has missing values. This makes out job quite easy. We will not throw away the missing observations. Instead, we will impute them using random forest. We don't have to do it manually. Instead, we will use `mice()` function from `mice` package.

```
set.seed(9009)

miceMod <- mice::mice(select(titanic_train,
                             -c(Survived, PassengerId, Name, Cabin, Ticket)),
                      method = "rf")  # perform mice imputation based on random forest.
```

```
##
##  iter imp variable
##    1   1  Age
##    1   2  Age
##    1   3  Age
##    1   4  Age
##    1   5  Age
##    2   1  Age
##    2   2  Age
##    2   3  Age
##    2   4  Age
##    2   5  Age
##    3   1  Age
##    3   2  Age
##    3   3  Age
##    3   4  Age
##    3   5  Age
##    4   1  Age
##    4   2  Age
##    4   3  Age
##    4   4  Age
```

```
##    4    5  Age
##    5    1  Age
##    5    2  Age
##    5    3  Age
##    5    4  Age
##    5    5  Age
```

Build a complete data set and add back 4 variables that we removed previously. Also convert `Survived` into a factor with more explicit labels.

```
titanic_train2 <- mice::complete(miceMod) %>%
  mutate(Name = titanic_train$Name,
         Cabin = titanic_train$Cabin,
         Tiket = titanic_train$Ticket,
         Survived = factor(ifelse(titanic_train$Survived == 1,
                                  "Survived", "Diseased")))
```

Check whether there are any missing values

```
anyNA(titanic_train2)
```

```
## [1] FALSE
```

There are no missing values any more.

Note that we did not use `Name` and `Cabin` to impute missing `age` because there is likely to be little information in these variables. But, interestingly, `Name` also contains the person's title, which can be extracted and used for model building. It can be a relevant variable in particular if it contains information that is not captured by other variables. I am refraining from doing it in order to keep this exercise short. Furthermore, it seems that adding these variables doesn't materially improve prediction accuracy. This is probably because these variables are associated with `Pclass`, `Sex`, and `Fare`.[1]

2.2 Training and test sets

Although Kaggle provided us with both training and test sets, we can't actually use the test set for model evaluation. Therefore, we must create our own test set. We will use `createDataPartition()` from `caret` package to create an index of row numbers to keep in the training set. The rest will go in the test set.

[1]There are several solutions to Titanic contest online. You can check their code to see how they used these variables in their model.

```
set.seed(5555)
index <- caret::createDataPartition(
  titanic_train2$Survived,
  p = 0.8,
  list = FALSE # Caret returns a list by default
)
```

```
t_train <- titanic_train2[index,]
t_test <- titanic_train2[-index,]
```

The great aspect of `createDataPartition` is that it keeps the proportion of the classes in the specified variables the same in the two data sets. Let's take a look:

```
table(titanic_train2$Survived) / length(titanic_train2$Survived)
```

```
##
##  Diseased  Survived
## 0.6161616 0.3838384
```

```
table(t_train$Survived) / length(t_train$Survived)
```

```
##
##  Diseased  Survived
## 0.6162465 0.3837535
```

```
table(t_test$Survived) / length(t_test$Survived)
```

```
##
##  Diseased  Survived
## 0.6158192 0.3841808
```

2.3 Logistic regression

We will first use binary logistic regression for model building using `t_train` data and then assess its performance using `t_test` data. For this we will use `caret` package although base R has `glm()` function which will also suffice.

```
m1 <- caret::train(Survived ~ .,
                   data = t_train[, -c(8:10)], # Drop Name, Ticket, and Cabin
                   method = "glm",
                   family = binomial())

summary(m1)
```

```
##
## Call:
## NULL
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -2.6753  -0.6524  -0.4155   0.6583   2.4333
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)  5.484932   0.649749   8.442  < 2e-16 ***
## Pclass      -1.170724   0.166886  -7.015 2.30e-12 ***
## Sexmale     -2.485338   0.219876 -11.303  < 2e-16 ***
## Age         -0.045550   0.008486  -5.367 7.99e-08 ***
## SibSp       -0.418137   0.119379  -3.503 0.000461 ***
## Parch       -0.099356   0.133415  -0.745 0.456445
## Fare         0.002014   0.002562   0.786 0.431715
## EmbarkedQ    0.105943   0.415548   0.255 0.798764
## EmbarkedS   -0.361276   0.262003  -1.379 0.167925
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 950.86  on 713  degrees of freedom
## Residual deviance: 644.08  on 705  degrees of freedom
## AIC: 662.08
##
## Number of Fisher Scoring iterations: 5
```

If you have seen the movie Titanic, perhaps you know that the ship's captain followed certain rules for evacuation. Women and children got to go first, and therefore, had a very high chance of survival. On the other hand, men from 3rd class had almost no chance of survival.

We get to see that playing out in the data. As `Pclass` increases, probability of survival drops. We will have to compute the odds ration to quantify this. Similarly, males on average had much smaller chance of survival compared to females. Next, younger passenger had a much higher chance of survival compared to an older passenger. Interestingly, people with siblings and/or spouse on Titanic had lower probability of survival! I don't know the reason for this.

2.3.1 Model performance

Let's check out the performance of the model out of the sample using `t_test` data set. For this, we will use `confusionMatrix()` function from `caret` package.

```
caret::confusionMatrix(predict(m1, subset(t_test, select = -Survived)),
                       reference = t_test$Survived,
                       positive = "Survived")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction Diseased Survived
##   Diseased       95       15
##   Survived       14       53
##
##                Accuracy : 0.8362
##                  95% CI : (0.7732, 0.8874)
##     No Information Rate : 0.6158
##     P-Value [Acc > NIR] : 1.38e-10
##
##                   Kappa : 0.6528
##
##  Mcnemar's Test P-Value : 1
##
##             Sensitivity : 0.7794
##             Specificity : 0.8716
##          Pos Pred Value : 0.7910
##          Neg Pred Value : 0.8636
##              Prevalence : 0.3842
##          Detection Rate : 0.2994
##    Detection Prevalence : 0.3785
##       Balanced Accuracy : 0.8255
##
##        'Positive' Class : Survived
##
```

It's a simple model and yet quite good! In most cases people were getting accuracies in low 80%s so we are not doing bad at all.[2]

2.4 XGBoost

The next model that we will consider is XGBoost. We will first set training controls. For more information please read `caret` documentation.

We willl opt for 5-fold cross-validation. Feel free to change the number to 10 if you want. Note that we are requesting class probabilities by setting `classProbs` to `TRUE`. This will return class

[2]Extension for you to try: Using interactions between variables (e.g., `Sex` and `Age`), check whether you can improve the accuracy of the model.

probabilities along with the predictions. Although we won't use these probabilities in this example, you can take a look at the predicted probabilities.[3] Setting `allowParallel` to `TRUE` will enable us to make use of parallel processing if you use a multi-core processor.

```
trControl <- trainControl(method = "cv",
                          number = 5,
                          verboseIter = FALSE,
                          classProbs = TRUE,
                          summaryFunction = twoClassSummary,
                          savePredictions = TRUE,
                          allowParallel = TRUE)
```

2.4.1 Hyperparameter tuning

Next, we will create a hyperparameter tuning grid. Hyperparameters are specific to a model. For instance, in the logistic regression, there is no hyperparameter to tune. However, in most machine learning techniques there will be hyperparameters and we have to find their optimal levels. Usually, the preferred method for that is grid search because the model is far too complex to have a closed form.

For XGBoost tree, the important hyperparameters to tune are as follows:

`eta`(η): This is also known as the learning rate. `eta` shrinks the weights associated with features/variables so this is a regularization parameter. $\eta \in [0, 1]$

`gamma` (γ): This is the minimum loss reduction that is required for further partitioning a leaf node. Thus, larger values of `gamma` are going to make model more conservative. $\gamma \in [0, \infty]$

`max_depth`: Maximum depth of a tree. A deeper tree is more complex and might overfit.

`min_child_weight`: From XGBoost documentation[4] - *Minimum sum of instance weight (hessian) needed in a child. If the tree partition step results in a leaf node with the sum of instance weight less than min_child_weight, then the building process will give up further partitioning. In linear regression task, this simply corresponds to minimum number of instances needed to be in each node. The larger* `min_child_weight` *is, the more conservative the algorithm will be.* `min_child_weight` $\in [0, \infty]$

`colsample_bytree`: The parameter that determines subsampling of variables. `colsample_bytree` $\in [0, 1]$

`subsample`: This is the percentage of observations to be used for training in each boosting iteration. The default is 1.

`nrounds`: This controls the maximum number of iterations. For classification, this is equivalent to the number of trees to grow.

```
tuneGrid <- expand.grid(nrounds = seq(10, 100, 10),
                        max_depth = seq(2, 8, 1),
```

[3] We use the probabilities in the Insurance Calls example: 4.8
[4] https://xgboost.readthedocs.io/en/latest/parameter.html

Table 2.3: Best Hyperparameters

	nrounds	max_depth	eta	gamma	colsample_bytree	min_child_weight	subsample
2153	30	2	0.2	0.1	1	1	1

```
                        eta = c(0.1, 0.2, 0.3),
                        gamma = 10^c(-1:3),
                        colsample_bytree = seq(0, 1, 0.2),
                        min_child_weight = 1,
                        subsample = 1)
```

2.4.2 Model training

The next piece of code will do the model training using the controls and grid we creates. Note that `tuneGrid` object has 6,300 rows, meaning that the model will be estimated 6,300 times. However, that's not the end of it. We also specify 5-fold cross-validation, which means the model will actually be estimated for 31,500 times! So this will likely take a lot of time. I strongly recommend not doing this in the class. To speed up the model execution, we will opt for parallel processing. For this we will use `doParallel` package. In the code below, input the number of cores you want to use for parallel processing. This will depend on your computer.

Warning: This code might take several minutes to execute depending on your computer!

```
# Don't run this code in the class

cl <- makePSOCKcluster(6)
registerDoParallel(cl)

set.seed(888)

m2 <- train(Survived ~. ,
            data = t_train[, -c(8:10)], # Drop Name, Ticket, and Cabin
            method = 'xgbTree',
            trControl = trControl,
            tuneGrid = tuneGrid)

stopCluster(cl) # Turn off parallel processing and free up the cores.
registerDoSEQ()
```

The best hyper parameters for this model and data appear to be as follows:

```
print(m2$bestTune)
```

Next we will use these parameters to build the model in the class.

```
m3 <- train(Survived ~. ,
            data = t_train[, -c(8:10)], # Drop Name, Ticket, and Cabin
            method = 'xgbTree',
            trControl = trControl,
            tuneGrid = data.frame(nrounds = 30,
                                  max_depth = 2,
                                  eta = 0.2,
                                  gamma = 0.1,
                                  colsample_bytree = 1,
                                  min_child_weight = 1,
                                  subsample = 1))
```

```
## Warning in train.default(x, y, weights = w, ...): The metric "Accuracy" was
## not in the result set. ROC will be used instead.
```

2.4.3 Model performance

```
confusionMatrix(predict(m3, subset(t_test, select = -Survived)),
                reference = t_test$Survived,
                positive = "Survived")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction Diseased Survived
##   Diseased      103       19
##   Survived        6       49
##
##                Accuracy : 0.8588
##                  95% CI : (0.7986, 0.9065)
##     No Information Rate : 0.6158
##     P-Value [Acc > NIR] : 9.459e-13
##
##                   Kappa : 0.6904
##
##  Mcnemar's Test P-Value : 0.0164
##
##             Sensitivity : 0.7206
##             Specificity : 0.9450
##          Pos Pred Value : 0.8909
##          Neg Pred Value : 0.8443
##              Prevalence : 0.3842
##          Detection Rate : 0.2768
##    Detection Prevalence : 0.3107
```

```
##       Balanced Accuracy : 0.8328
##
##        'Positive' Class : Survived
##
```

Turns out that XGBoost did only about as good as logistic regression. This just goes on to show that logistic regression in many cases is still a good algorithm to use.[5]

[5] What changes can you make to your logistic regression model so that it produces better predictions?

Chapter 3

Predicting Wine Quality

This chapter shows you how to deal with dependent variables that are categorical in nature and have more than two levels. Because the variable is not binary, the modeling becomes more complex. In this chapter you will learn how to use:

1. Multinomial logistic regression,
2. Support Vector Machines, and
3. Ordinal Regression

3.1 Task Description

For this task, we will use wine quality data set available here: https://archive.ics.uci.edu/ml/datasets/Wine+Quality. There are separate CSV files for white and red wine. Combine them and make a larger united file.

3.1.1 Data Set Information

This part is verbatim reproduced from UCI.

The two data sets are related to red and white variants of the Portuguese "Vinho Verde" wine. For more details, consult the paper by Cortez et al., 2009. Due to privacy and logistic issues, only physicochemical (inputs) and sensory (the output) variables are available (e.g. there is no data about grape types, wine brand, wine selling price, etc.).

These datasets can be viewed as classification or regression tasks. The classes are ordered and not balanced (e.g. there are much more normal wines than excellent or poor ones). Outlier detection algorithms could be used to detect the few excellent or poor wines. Also, we are not sure if all input variables are relevant. So it could be interesting to test feature selection methods. [1]

Attribute Information: For more information, read [Cortez et al., 2009]. Input variables (based on physicochemical tests):

1. fixed acidity

[1] We are not doing feature selection in this exercise.

2. volatile acidity
3. citric acid
4. residual sugar
5. chlorides
6. free sulfur dioxide
7. total sulfur dioxide
8. density
9. pH
10. sulphates
11. alcohol

Output variable (based on sensory data):

12. quality (score between 0 and 10)

3.2 Specific Tasks to Complete

1. Build a logistic regression model using **`quality`** as the target variable. Note that you don't have binary classification task any more. For this you will have to use multinomial logistic regression. However, you can still interpret the model output (i.e., statistical significance of the coefficients, etc. exactly the same way as binary logistic regression).
2. Use support vector machine (SVM) to estimate the model. Treat quality as a multiple categorical variable.
3. Treat quality as a continuous variable. Estimate a linear regression model and compare the output with multinomial regression and SVM.
4. Estimate the model using quality as an ordinal variable.

3.3 Data

Load all the relevant packages. If you do not have any of these packages installed, use `install.package()` function to install it from CRAN.

```
library(caret)
library(dplyr)
library(moments)
library(ggplot2)
```

```
library(ggcorrplot)
library(e1071)
library(doParallel)
library(nnet)
library(reshape2)
library(ordinal)
```

3.3.1 Read wine data

For this solution, I am going to read the data sets that I have already downloaded and saved on Github. you don't have to use these but if you want to, the data sets are available from my public Github repository

We can directly read the CSV files using `read.csv()` function from Base R. I have cleaned up the data a little bit. There are separate CSV files for red and white wine. First, we will read the two data files and add a column to indicate which type of wine it is. Finally, we will stack the two data sets on top of each other using `rbind()` function from Base R.

```
red <- read.csv("http://bit.ly/2LvaPv7",
                stringsAsFactors = FALSE) %>%
  mutate(wine = "red")

white <- read.csv("http://bit.ly/2VlYfCJ",
                  stringsAsFactors = FALSE) %>%
  mutate(wine = "white")

wine <- rbind(red, white) %>%
   mutate(wine = as.factor(wine))
```

Note that I changed the variable `class` of `wine` to **`factor`**. This is because it will be easier for us to use this variable directly in the models as R will internally create a indicator variable such that red wine will equal 0 and white wine will equal 1. We will later create a variable that deals with explicitly.

3.4 Summarize data

Start with basic summary using base R.

```
summary(wine)
```

```
##  fixed_acidity    volatile_acidity  citric_acid     residual_sugar
##  Min.   : 3.800   Min.   :0.0800   Min.   :0.0000   Min.   : 0.600
##  1st Qu.: 6.400   1st Qu.:0.2300   1st Qu.:0.2500   1st Qu.: 1.800
```

```
##  Median : 7.000   Median :0.2900   Median :0.3100   Median : 3.000
##  Mean   : 7.215   Mean   :0.3397   Mean   :0.3186   Mean   : 5.443
##  3rd Qu.: 7.700   3rd Qu.:0.4000   3rd Qu.:0.3900   3rd Qu.: 8.100
##  Max.   :15.900   Max.   :1.5800   Max.   :1.6600   Max.   :65.800
##    chlorides       free_sulfur_dioxide total_sulfur_dioxide
##  Min.   :0.00900   Min.   :  1.00      Min.   :  6.0
##  1st Qu.:0.03800   1st Qu.: 17.00      1st Qu.: 77.0
##  Median :0.04700   Median : 29.00      Median :118.0
##  Mean   :0.05603   Mean   : 30.53      Mean   :115.7
##  3rd Qu.:0.06500   3rd Qu.: 41.00      3rd Qu.:156.0
##  Max.   :0.61100   Max.   :289.00      Max.   :440.0
##     density             pH          sulphates         alcohol
##  Min.   :0.9871   Min.   :2.720   Min.   :0.2200   Min.   : 8.00
##  1st Qu.:0.9923   1st Qu.:3.110   1st Qu.:0.4300   1st Qu.: 9.50
##  Median :0.9949   Median :3.210   Median :0.5100   Median :10.30
##  Mean   :0.9947   Mean   :3.219   Mean   :0.5313   Mean   :10.49
##  3rd Qu.:0.9970   3rd Qu.:3.320   3rd Qu.:0.6000   3rd Qu.:11.30
##  Max.   :1.0390   Max.   :4.010   Max.   :2.0000   Max.   :14.90
##     quality         wine
##  Min.   :3.000   red  :1599
##  1st Qu.:5.000   white:4898
##  Median :6.000
##  Mean   :5.818
##  3rd Qu.:6.000
##  Max.   :9.000
```

A few of these variables have very tight distributions (e.g., `density`). Also, extreme values might cause a problem in some other cases (e.g., `residual_sugar`). We will have to correct these later on.

Our dependent variable is `quality`. As we will be using it as a categorical variable in 3 of the 4 models, let's look at its frequency distribution, which we did not get from `summary()` function because `quality` is not categorical.

```
table(wine$quality)
```

```
##
##    3    4    5    6    7    8    9
##   30  216 2138 2836 1079  193    5
```

Clearly, the categories at the extremes have very few observations. This will lead to problems in correctly categorizing extreme values. In order to overcome this problem, we will create two new variables.

3.5 Create new variants of `quality`

For support vector machines (SVM) and multinomial logistic model (MNL), we will create a new variable labeled `quality.c` which will be a factor variable with groups 3, 4, 8, and 9 combined in another group. We can label this combined group anything we want as the labeling is meaningless for these methods. I will label this new group `3489`, thereby preserving the knowledge that this group came from 4 separate groups. For many operations, `caret` package requires that factor levels should be valid variable names. Therefore, we will add a prefix `q_` before the numbers in `quality` to create `quality.c`.

For ordinal regression, we have a little bit more information about the ordering of the groups. We will create a new variable `quality.o`. In this variable, we will combine 3, 4, and 5 and label it 5 to indicate this as "5 and lower". Similarly, we will combine 7, 8, and 9 and label it 7 to indicate that this group is "7 and above". Thus, we will effectively have only 3 groups.

Clearly, this will make model comparison a little bit tough but we have to give each model the best chance to perform even at this lower level of analysis.

```
wine <- wine %>%
  mutate(quality.c = ifelse(quality %in% c(5, 6, 7),
                            paste0("q_", quality),
                            "q_3489"),
         quality.o = ifelse(quality <= 5,
                            5,
                            ifelse(quality >= 7, 7, 6))) %>%
  mutate(quality.c = factor(quality.c,
                            levels = c("q_5", "q_6",
                                       "q_7", "q_3489"))) %>%
  mutate(quality.o = ordered(quality.o))
```

Check the structure of new variables.

```
str(wine$quality.c)
```

```
##  Factor w/ 4 levels "q_5","q_6","q_7",..: 1 1 1 2 1 1 1 3 3 1 ...
```

```
str(wine$quality.o)
```

```
##  Ord.factor w/ 3 levels "5"<"6"<"7": 1 1 1 2 1 1 1 3 3 1 ...
```

3.6 Predictor variables

Now that we have new dependent variables, let's take a look at the predictor variables. As we are doing a predictive analysis (we have no plan to do a statistical inference), let's understand the distribution and correlations of the variables and explore the need for transformation.

Table 3.1: Correlation Coefficients

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11
V1 : fixed_acidity	1										
V2 : volatile_acidity	0.22	1									
V3 : citric_acid	0.32	-0.38	1								
V4 : residual_sugar	-0.11	-0.2	0.14	1							
V5 : chlorides	0.3	0.38	0.04	-0.13	1						
V6 : free_sulfur_dioxide	-0.28	-0.35	0.13	0.4	-0.2	1					
V7 : total_sulfur_dioxide	-0.33	-0.41	0.2	0.5	-0.28	0.72	1				
V8 : density	0.46	0.27	0.1	0.55	0.36	0.03	0.03	1			
V9 : pH	-0.25	0.26	-0.33	-0.27	0.04	-0.15	-0.24	0.01	1		
V10 : sulphates	0.3	0.23	0.06	-0.19	0.4	-0.19	-0.28	0.26	0.19	1	
V11 : alcohol	-0.1	-0.04	-0.01	-0.36	-0.26	-0.18	-0.27	-0.69	0.12	0	1
V12 : quality	-0.08	-0.27	0.09	-0.04	-0.2	0.06	-0.04	-0.31	0.02	0.04	0.44

For this we will first get the descriptive statistics and correlations for all the numeric variables. Table 3.1 shows the correlations.

```
cormat <- round(cor(as.matrix(wine[,-c(13,14,15)])),2)
cormat[upper.tri(cormat)] <- ""
cormat <- as.data.frame(cormat) %>% select(-quality)
colnames(cormat) <- c("V1", "V2", "V3", "V4", "V5",
                      "V6", "V7", "V8", "V9", "V10", "V11")
rownames(cormat) <- paste(c(colnames(cormat), "V12"),
                          ":",
                          rownames(cormat))
print(cormat)
```

Next we will use `ggcorrplot` package to create a nice looking correlation plot. This package is available on CRAN

```
ggcorrplot::ggcorrplot(round(cor(as.matrix(wine[, -c(13,14,15)])), 2),
            p.mat = ggcorrplot::cor_pmat(as.matrix(wine[, -c(13,14,15)])),
            hc.order = TRUE, type = "lower",
            outline.col = "white",
            ggtheme = ggplot2::theme_minimal,
            colors = c("#cf222c", "white", "#3a2d7f")
            )
```

In the above heat map, the crosses indicate non-significant correlations. From the correlations, most variables have their own unique information set. However, it appears that `quality` is strongly related to only a few variables. This is not great news![2]

[2]At this point one can think of transformations to increase the correlations between the variables. However, I am not going to do it as this will make this exercise broader than I want. This is left to the reader as an exercise.

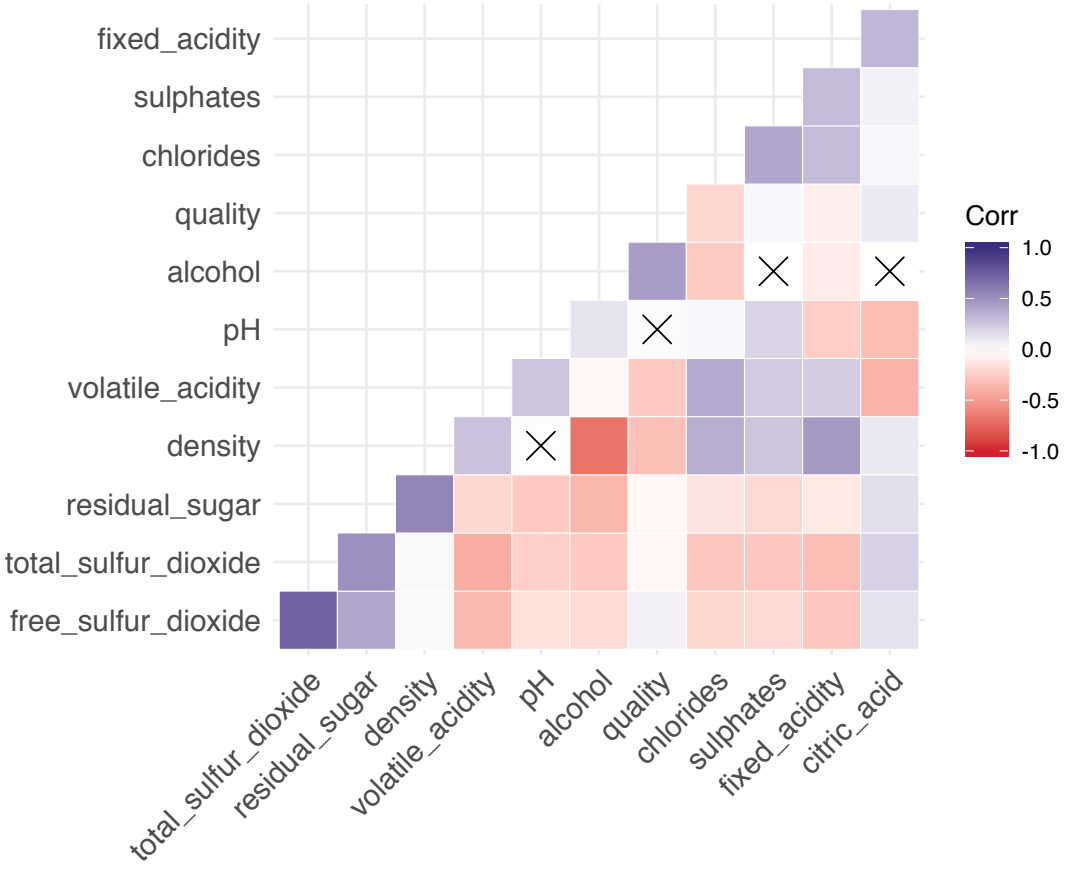


Figure 3.1: Correlation Heatmap

Table 3.2: Detailed Summary Statistics

	Mean	Median	Std_Dev	CV	Skewness	Kurtosis
fixed_acidity	7.22	7.00	1.30	0.18	1.72	5.05
volatile_acidity	0.34	0.29	0.16	0.48	1.49	2.82
citric_acid	0.32	0.31	0.15	0.46	0.47	2.39
residual_sugar	5.44	3.00	4.76	0.87	1.43	4.35
chlorides	0.06	0.05	0.04	0.63	5.40	50.84
free_sulfur_dioxide	30.53	29.00	17.75	0.58	1.22	7.90
total_sulfur_dioxide	115.74	118.00	56.52	0.49	0.00	-0.37
density	0.99	0.99	0.00	0.00	0.50	6.60
pH	3.22	3.21	0.16	0.05	0.39	0.37
sulphates	0.53	0.51	0.15	0.28	1.80	8.64
alcohol	10.49	10.30	1.19	0.11	0.57	-0.53
quality	5.82	6.00	0.87	0.15	0.19	0.23

3.7 More descriptive statistics

Let's get more descriptive statistics in order to understand the distribution of our variables a little better.

```
wine_temp <- wine[,-c(13,14,15)]

desc <- as.data.frame(cbind(Mean = sapply(wine_temp, mean),
                      Median = sapply(wine_temp, median),
                      Std_Dev = sapply(wine_temp, sd),
                      CV = sapply(wine_temp, sd) / sapply(wine_temp, mean),
                      Skewness = sapply(wine_temp, skewness),
                      Kurtosis = sapply(wine_temp, kurtosis)))
```

```
round(desc,2)
```

The most interesting column for me is the CV (coefficient of variation). This is the ratio of standard deviation to the mean. We have certain observations where CV is very low (e.g., 0 or 0.05). This means that the standard deviation is extremely small compared to the mean. Clearly we need some scaling here to remove the effect of the mean. One way to do that is to mean center all the variables so that we have zero mean all across. It retains the low standard deviation, however. To overcome this issue, we can divide all the variables by their standard deviations. This way, we will normalize our data such that all the variables will have mean = 0 and standard deviation = 1.[3]

Let's scale the numeric variables.

[3]Will that help with reducing skewness and kurtosis? Think about it for a moment before you read on.

Table 3.3: Summary Statistics of Scales Variables

	Mean	Median	Std.Dev	Skewness	Kurtosis
fixed_acidity	0	-0.17	1	1.72	5.05
volatile_acidity	0	-0.30	1	1.49	2.82
citric_acid	0	-0.06	1	0.47	2.39
residual_sugar	0	-0.51	1	1.43	4.35
chlorides	0	-0.26	1	5.40	50.84
free_sulfur_dioxide	0	-0.09	1	1.22	7.90
total_sulfur_dioxide	0	0.04	1	0.00	-0.37
density	0	0.06	1	0.50	6.60
pH	0	-0.05	1	0.39	0.37
sulphates	0	-0.14	1	1.80	8.64
alcohol	0	-0.16	1	0.57	-0.53
quality	0	0.21	1	0.19	0.23

```
# First create a duplicate dataset
wine2 <- wine
wine2[,c(1:12)] <- scale(wine[ , c(1:12)])

desc2 <- as.data.frame(cbind(Mean = sapply(wine2[ , c(1:12)], mean),
                      Median = sapply(wine2[ , c(1:12)], median),
                      Std.Dev = sapply(wine2[ , c(1:12)], sd),
                      Skewness = sapply(wine2[ , c(1:12)], skewness),
                      Kurtosis = sapply(wine2[ , c(1:12)], kurtosis)))
```

```
round(desc2,2)
```

Scaling doesn't affect skewness or kurtosis. In order to alter these two moments, we need to use nonlinear transformation such as logarithmic or square root transformations. I'm going to do it through trial and error.

Normal distribution has skewness = 0 and kurtosis = 3. `total_sulfur_dioxide`, `pH`, `alcohol`, and `quality` seem to have this shape (a good idea is to plot these distributions). I am concerned about `fixed_acidity`, `chlorides`, `free_sulfur_dioxide`, `density`, and `sulphates` due to high kurtosis (and skewness in some cases). Let's take their log transform first and then scale these variables.

```
wine2[ , c(1, 5, 6, 8, 10)] <- scale(log(wine[ , c(1, 5, 6, 8, 10)]))
```

Print skewness and kurtosis.

```
moments::skewness(wine2[,c(1, 5, 6, 8, 10)])
```

Table 3.4: Skewness

fixed_acidity	0.8889319
chlorides	0.8762698
free_sulfur_dioxide	-0.8340045
density	0.4672599
sulphates	0.4048986

Table 3.5: Kurtosis

fixed_acidity	4.896783
chlorides	5.305355
free_sulfur_dioxide	3.429675
density	9.008338
sulphates	3.701296

```
moments::kurtosis(wine2[,c(1, 5, 6, 8, 10)])
```

Except for `density` the remaining 4 variables benefited from log transformation. After plotting the distribution for `density` it appears that this might be because of a couple of extreme values. Although log transformation should have gotten rid of them, it seems it didn't work out. So I replaced the two extreme values (there were 3 observations) with the next highest observation and then took log. As it turns out, the transformation paid off.

```
# replace the values > 1.00369 by 1.00369

wine2$density <- ifelse(wine$density > 1.00369,
                        1.00369,
                        wine$density)
wine2$density <- scale(log(wine2$density))

moments::skewness(wine2$density)
```

```
## [1] -0.02258351
```

```
moments::kurtosis(wine2$density)
```

```
## [1] 2.254121
```

Now we have a data set `wine2` which has all the transformed variables. We will use it for the rest of the analysis.

3.8 Linear regression model

For linear regression model, our assumption is that `quality` is a continuous variable. It's certainly a debatable assumption. However, in many practical cases, a linear regression model works out

pretty OK so I am starting off with it.

```
model.lm <- caret::train(quality ~ .,
                         data = wine2[, -c(14,15)],
                         method = "lm")
summary(model.lm)
```

```
##
## Call:
## lm(formula = .outcome ~ ., data = dat)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -3.8267 -0.5347 -0.0440  0.5187  3.4109
##
## Coefficients:
##                       Estimate Std. Error t value Pr(>|t|)
## (Intercept)           0.317675   0.051352   6.186 6.54e-10 ***
## fixed_acidity         0.123372   0.023010   5.362 8.54e-08 ***
## volatile_acidity     -0.275112   0.015059 -18.269  < 2e-16 ***
## citric_acid          -0.004644   0.012951  -0.359   0.7199
## residual_sugar        0.304732   0.031516   9.669  < 2e-16 ***
## chlorides            -0.039373   0.015838  -2.486   0.0129 *
## free_sulfur_dioxide   0.194606   0.016048  12.126  < 2e-16 ***
## total_sulfur_dioxide -0.155376   0.020783  -7.476 8.65e-14 ***
## density              -0.317525   0.050289  -6.314 2.90e-10 ***
## pH                    0.080127   0.016593   4.829 1.40e-06 ***
## sulphates             0.120871   0.012814   9.433  < 2e-16 ***
## alcohol               0.305010   0.025422  11.998  < 2e-16 ***
## winewhite            -0.421383   0.066724  -6.315 2.87e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.8328 on 6484 degrees of freedom
## Multiple R-squared:  0.3077, Adjusted R-squared:  0.3065
## F-statistic: 240.2 on 12 and 6484 DF,  p-value: < 2.2e-16
```

We get a decent model with adjusted-R^2 = 0.3065. The model is highly significant with p-value of F-statistic < 2.2E-16. Except for `citric_acid` all the other variables are statistically significant at conventional levels.

Note that I also included the variable `wine` in the data set while estimating the model. As it turns out, white wines have on average lower rating than red wines (all else equal).[4]

[4] I would have liked to tweak this model a little bit to understand if there are any interactions present. I will leave it for the readers as an exercise.

3.9 Multinomial logistic regression (MNL)

For MNL, we will use `quality.c` as the dependent variable. Recall that this is a categorical variable with groups 3, 4, 8, and 9 bundled together.[5]

```
table(wine2$quality.c)
```

```
##
##    q_5    q_6    q_7 q_3489
##   2138   2836   1079    444
```

We will use `caret` to estimate MNL using its `multinom` method. Note that `caret` uses `nnet` (CRAN) under the hood for estimating MNL.

MNL does not require a tuning parameter. However, if we want to try out a penalized (regularized) MNL, the tuning parameter is `decay` ($decay \in [0, 1]$).[6]

We will start with defining the train control and creating a parameter grid.

```
trControl_mnl <- trainControl(method = "cv",
                              number = 10,
                              search = "grid",
                              classProbs = TRUE,
                              summaryFunction = multiClassSummary)

tuneGrid_mnl <- expand.grid(decay = seq(0, 1, by = 0.1))
```

Note that we are using `multiClassSummary()` for `summaryFunction` argument. This will return very detailed summary for all the classes, which can be useful when we use `confusionMatrix()` function.

Before we estimate MNL, we need to create a train and test set. The next code chunk creates a 80% training set and 20% test set.

```
set.seed(2091)
index <- caret::createDataPartition(wine2$quality.c,
                                    p = 0.8,
                                    list = FALSE)

train_wine <- wine2[index, ]
test_wine <- wine2[-index, ]
```

[5] Before we even run any model, I can tell you that this bundling is going to cause some problems. This is because we have assumed no ordering in `quality`. But in reality there is an ordering: 9 > 8 > 7 > ...> 3. We will repeat this analysis with `quality.o` later with ordinal logit.

[6] In the documentation for `nnet` I could not find out whether the regularization is LASSO or Ridge.

Now we are ready to estimate MNL. MNL is a parametric model that is commonly estimated using maximum likelihood estimation. As the likelihood function does not have a closed form, likelihood is maximized using an iterative process. We can provide maximum iterations to use for estimating the model, which we set at 100. Usually `multinom` displays the outcome of every 10th iterations. In our case, we specified 10-fold CV and 11 potential values for `decay`. This means that we will have 110 outputs showing results of 10 iteration blocks! We probably do not need so much output so we can turn it off by specifying `trace = FALSE`.

The following code may take some time to run. On my Mac it took about 28 seconds.

```
model_mnl <- caret::train(quality.c ~ .,
                          data = train_wine[ , -c(12,15)],
                          method = "multinom",
                          maxit = 100,
                          trace = FALSE, # suppress iterations
                          tuneGrid = tuneGrid_mnl,
                          trControl = trControl_mnl
                          )
```

You might get a warning that there were missing values in resampled performance measures. Unless you really care about all the performance metrics that `caret` reports, this is not a major concern.

Let's take a look at the best value for `decay`. For this model it is 0.2.

```
model_mnl$bestTune
```

```
##   decay
## 3   0.2
```

If you want to take a look at the full summary for each value of `decay`, we can print `model_mnl$results` data frame. Below, I print only AUC and Accuracy. It seems that the accuracy is not sensitive to the decay.

```
model_mnl$results %>%
   select(decay, AUC, Accuracy)
```

```
##    decay       AUC  Accuracy
## 1    0.0 0.7210105 0.5413473
## 2    0.1 0.7209793 0.5423092
## 3    0.2 0.7208119 0.5423092
## 4    0.3 0.7207245 0.5421173
## 5    0.4 0.7206714 0.5417323
## 6    0.5 0.7205310 0.5417327
## 7    0.6 0.7204086 0.5415404
## 8    0.7 0.7203053 0.5413488
```

```
## 9    0.8 0.7201930 0.5411565
## 10   0.9 0.7200828 0.5407715
## 11   1.0 0.7199869 0.5407711
```

3.9.1 Model parameters

We have to calculate z-statistics and p-values manually as `multinom` does not report them. We first create an object that holds the summary of `model_mnl`. Next, we calculate z-statistics by dividing model coefficients by standard errors. Finally, we calculate p-values for 2-tailed test of significance of model parameters.

```
sum.mnl.c <- summary(model_mnl)
z <- sum.mnl.c$coefficients / sum.mnl.c$standard.errors
p <- (1 - pnorm(abs(z), 0, 1)) * 2
```

To make the coefficients easy to read, let's create 3 data frames that will hold all the coefficients, standard errors, z statistics, and p values. Note that MNL uses one of the categories as reference levels. If we do not explicitly specify it, it will consider the first category, in our case `quality = 5`, as the reference category. So all the results must be interpreted with respect to the reference level.

```
# Coefficients for quality = 6

coeff.mnl.c1 <- rbind(sum.mnl.c$coefficients[1,],
                      sum.mnl.c$standard.errors[1,],
                      z[1,],
                      p[1,])

rownames(coeff.mnl.c1) <- c("Coefficient", "Std. Errors",
                            "z stat", "p value")

# Coefficients for quality = 7

coeff.mnl.c2 <- rbind(sum.mnl.c$coefficients[2,],
                      sum.mnl.c$standard.errors[2,],
                      z[2,],
                      p[2,])

rownames(coeff.mnl.c2) <- c("Coefficient", "Std. Errors",
                            "z stat", "p value")

# Coefficients for quality = 3489

coeff.mnl.c3 <- rbind(sum.mnl.c$coefficients[3,],
                      sum.mnl.c$standard.errors[3,],
```

Table 3.6: Coefficients for Wine Quality = 6

	Coefficient	Std. Errors	z stat	p value
(Intercept)	0.69	0.18	3.86	0.00
fixed_acidity	0.08	0.08	0.94	0.35
volatile_acidity	-0.56	0.05	-10.12	0.00
citric_acid	-0.03	0.04	-0.76	0.45
residual_sugar	0.41	0.11	3.64	0.00
chlorides	0.01	0.05	0.23	0.82
free_sulfur_dioxide	0.22	0.05	4.01	0.00
total_sulfur_dioxide	-0.35	0.07	-5.05	0.00
density	-0.28	0.18	-1.57	0.12
pH	0.12	0.06	2.05	0.04
sulphates	0.26	0.05	5.62	0.00
alcohol	0.82	0.09	8.78	0.00
winewhite	-0.16	0.23	-0.68	0.50

Table 3.7: Coefficients for Wine Quality = 7

	Coefficient	Std. Errors	z stat	p value
(Intercept)	-0.45	0.25	-1.80	0.07
fixed_acidity	0.52	0.11	4.62	0.00
volatile_acidity	-0.92	0.09	-10.72	0.00
citric_acid	-0.08	0.07	-1.15	0.25
residual_sugar	0.94	0.15	6.08	0.00
chlorides	-0.24	0.08	-2.95	0.00
free_sulfur_dioxide	0.38	0.08	4.84	0.00
total_sulfur_dioxide	-0.53	0.10	-5.20	0.00
density	-0.93	0.25	-3.73	0.00
pH	0.37	0.08	4.65	0.00
sulphates	0.52	0.06	8.60	0.00
alcohol	1.24	0.13	9.70	0.00
winewhite	-0.57	0.33	-1.74	0.08

```
                      z[3,],
                      p[3,])

rownames(coeff.mnl.c3) <- c("Coefficient", "Std. Errors",
                            "z stat", "p value")
```

```
quality.c = 6
```

```
quality.c = 7
```

```
quality.c = 3489
```

It's a good exercise to compare the coefficients of MNL and linear regressions. However, MNL has 3 equations with different variables showing up significant across the 3 models. What can you do now? One way to handle this problem is to first identify the variables that are significant

Table 3.8: Coefficients for Wine Quality = 3489

	Coefficient	Std. Errors	z stat	p value
(Intercept)	-3.30	0.33	-10.12	0.00
fixed_acidity	0.32	0.14	2.37	0.02
volatile_acidity	0.33	0.08	4.45	0.00
citric_acid	-0.06	0.08	-0.73	0.46
residual_sugar	0.25	0.17	1.46	0.14
chlorides	0.04	0.09	0.47	0.64
free_sulfur_dioxide	-0.26	0.09	-2.99	0.00
total_sulfur_dioxide	-0.20	0.12	-1.67	0.10
density	-0.30	0.28	-1.06	0.29
pH	0.30	0.10	3.08	0.00
sulphates	0.09	0.08	1.09	0.28
alcohol	0.88	0.15	5.96	0.00
winewhite	2.35	0.40	5.89	0.00

across the board. We will retain them. Then look at the ones that are non-significant in all of the 3 equations. Most likely these variables can be dropped from the next iteration of MNL. The variables that are significant in a few equations are the most difficult ones. You will have to take a call depending on their overall importance. If you drop them and the performance of the model deteriorates severely, perhaps adding them back is a better choice. You can also be conservative and keep any variable that's significant at least once in the 3 models.

3.9.2 Model performance

MNL model performance can be assessed on several different metrics. I will select classification accuracy as the relevant metric. As such let's get the confusion matrix by using the same samples that we used for estimating the model. You could do this using cross-validation.

```
caret::confusionMatrix(predict(model_mnl,
                               newdata = test_wine[, -c(12, 15)],
                               type = "raw"),
                       reference = test_wine$quality.c)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction q_5 q_6 q_7 q_3489
##     q_5    256 130  17     27
##     q_6    165 390 166     46
##     q_7      4  43  32     11
##     q_3489   2   4   0      4
##
## Overall Statistics
##
```

```
##                Accuracy : 0.5258
##                  95% CI : (0.4982, 0.5533)
##     No Information Rate : 0.4372
##     P-Value [Acc > NIR] : 8.896e-11
##
##                   Kappa : 0.2356
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
## Statistics by Class:
##
##                      Class: q_5 Class: q_6 Class: q_7 Class: q_3489
## Sensitivity              0.5995     0.6878    0.14884      0.045455
## Specificity              0.8000     0.4836    0.94640      0.995037
## Pos Pred Value           0.5953     0.5085    0.35556      0.400000
## Neg Pred Value           0.8028     0.6660    0.84838      0.934732
## Prevalence               0.3292     0.4372    0.16577      0.067849
## Detection Rate           0.1974     0.3007    0.02467      0.003084
## Detection Prevalence     0.3315     0.5914    0.06939      0.007710
## Balanced Accuracy        0.6998     0.5857    0.54762      0.520246
```

The model accuracy is 54%, which is not too bad. However, kappa is just 0.26 suggesting that the model is performing poorly on this metric.[7] Note that class 6 has a prevalence of around 44%. Thus, you would be right 44% of the times if you labeled all wines as 6. However, the model did a particularly poor job of detecting classes 7 and 3489. But this is the first MNL model you have estimated. You should tweak this model further to yield better results.

3.10 Support Vector Machines (SVM)

The advantage of using SVM is that although it is a linear model, we can use kernels to model linearly non-separable data. We will use the default radial basis function (RBF) kernel for SVM. An SVM with RBF takes two hyper parameters that we need to tune before estimating SVM. But it takes a long time to tune. Therefore, in this example I won't actually tune it because I have already done it previously using `e1071` package.

We will use `e1071` and `caret` separately to get SVM.

Use the following code to estimate SVM using `e1071` package.

3.10.1 Hyperparameters tuning

For SVM with RBF, we need to tune `gamma` and `cost` hyperparameters. Whereas `cost` is generic to SVM with any kernel function, `gamma` is specific to RBF kernel, which is given as follows:

[7] Usually kappa < 0.3 is considered poor.

$$K(x_i, x_j) = e^{(-\gamma||x_i - x_j||^2)}$$

where x_i are the sample points and x_j are the support vectors. γ controls the extent to which support vectors exert influence on classification of any sample point.[8]

Higher values of `gamma` will lead to higher in-sample error (high bias) and lower out-of-sample error (low variance).

`cost` decides the penalty we want to set for not classifying sample points correctly. Therefore, lower cost will lead to more accommodating behavior leading to higher in-sample error (high bias) and lower out-of-sample error (low variance).

```
# Use multiple cores on your computer
# Change the number of cores depending on your processer

doParallel::registerDoParallel(cores = 4)

set.seed(1234)
svm.tune <- tune.svm(quality.c ~ .,
                     data = train_wine[, -c(12,15)],
                     gamma = c(0.05, 0.1, 0.5, 1, 2),
                     cost = 10^(0:3))
```

Print the best `gamma` and `cost`

```
svm.tune$best.parameters
```

```
##    gamma cost
## 19     1 1000
```

We find that `gamma = 1` and `cost = 1000` give us the best accuracy. You can also print the table for the performance of the entire grid by running the following code. This is not executed for this example.

```
svm.tune$performances
```

3.10.2 Model estimation

Using the best hyperparameters we can estimate the model. In the next code block I will use the hyperparameters from the first pass. Note that the `summary()` function for `svm()` doesn't provide any model parameters.

[8] RBF is the Gaussian kernel and $\gamma = 1/\sigma^2$

```
model.svm.c <- e1071::svm(quality.c ~ .,
                          data = train_wine[, -c(12, 15)],
                          kernel = "radial",
                          gamma = 1,
                          cost = 1000)
```

Print the model summary. Note that the `summary()` function for `svm()` doesn't provide any model parameters.

```
summary(model.svm.c)
```

```
##
## Call:
## svm(formula = quality.c ~ ., data = train_wine[, -c(12, 15)],
##     kernel = "radial", gamma = 1, cost = 1000)
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  1000
##
## Number of Support Vectors:  4389
##
##  ( 1393 1899 751 346 )
##
##
## Number of Classes:  4
##
## Levels:
##  q_5 q_6 q_7 q_3489
```

We have 4,389 support vectors. This suggests that the model is *memorizing* rather than *learning* as the training data set has only 5,200 observations. This could be because of many factors but th emost critical factor is that the predictor variables do not have much information to predict the wine quality.

3.10.3 Assessing the model performance

```
caret::confusionMatrix(reference = test_wine$quality.c,
                       predict(model.svm.c,
                               newdata = test_wine[, -c(12, 14, 15)],
                               type = "class"))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction q_5 q_6 q_7 q_3489
##     q_5    398  13   2      3
##     q_6     26 543  13      6
##     q_7      0  10 199      2
##     q_3489   3   1   1     77
##
## Overall Statistics
##
##                Accuracy : 0.9383
##                  95% CI : (0.9238, 0.9508)
##     No Information Rate : 0.4372
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.9072
##
##  Mcnemar's Test P-Value : 0.1005
##
## Statistics by Class:
##
##                      Class: q_5 Class: q_6 Class: q_7 Class: q_3489
## Sensitivity              0.9321     0.9577     0.9256       0.87500
## Specificity              0.9793     0.9384     0.9889       0.99586
## Pos Pred Value           0.9567     0.9235     0.9431       0.93902
## Neg Pred Value           0.9671     0.9661     0.9853       0.99095
## Prevalence               0.3292     0.4372     0.1658       0.06785
## Detection Rate           0.3069     0.4187     0.1534       0.05937
## Detection Prevalence     0.3207     0.4534     0.1627       0.06322
## Balanced Accuracy        0.9557     0.9480     0.9572       0.93543
```

It looks like SVM classification on the data is far better than MNL! We get the model accuracy up to 67% from a mere 54% for MNL. Recall that the no information rate is only 44% so with SVM we could improve the accuracy by 23%.

3.10.4 SVM with `caret`

The following code will tune the model but it's actually not going to run in this tutorial because it will take a lot of time.

```
ctrl <- caret::trainControl(method = "cv",
                            number = 10)

set.seed(1492)
grid <- expand.grid(sigma = 10^(-1:4),
```

```
                    C = 10^(0:4))

model.svm.tune <- train(quality.c ~ .,
                        data = train_wine[ , -c(12, 15)],
                        method = "svmRadial",
                        tuneGrid = grid,
                        trControl = ctrl)
```

Print the best parameter combination.

```
model.svm.tune$bestTune
```

```
##   sigma   C
## 8     1 100
```

We will use `sigma = 1` and `cost = 100` and estimate the model. We will then use `varImp` in `caret` to get the variable importance. In order to plot variable importance, all the predictor variables have to be non-character. As we have a factor variable, `wine`, we should create a indicator dummy ourselves.

```
train_wine$white_wine = ifelse(train_wine$wine == "white", 1, 0)
test_wine$white_wine = ifelse(test_wine$wine == "white", 1, 0)
```

With the best parameters, estimate the model.

```
ctrl <- caret::trainControl(method = "cv",
                            number = 10)

grid <- expand.grid(sigma = 1, C = 100)

set.seed(89753)

model.svm.c1 <- train(quality.c ~ .,
                      data = train_wine[,-c(12, 13, 15)],
                      method = "svmRadial",
                      tuneGrid = grid,
                      trControl = ctrl)
```

```
print(model.svm.c1)
```

```
## Support Vector Machines with Radial Basis Function Kernel
##
## 5200 samples
##   12 predictor
```

```
##    4 classes: 'q_5', 'q_6', 'q_7', 'q_3489'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 4680, 4679, 4680, 4679, 4679, 4681, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.6426823  0.4405893
##
## Tuning parameter 'sigma' was held constant at a value of 1
##
## Tuning parameter 'C' was held constant at a value of 100
```

In-sample accuracy of the model is 64%. Next, we will check the number of support vectors. Similar to the SVM fitted using `e1071`, we get 4,386 support vectors using `caret`, which underneath uses `kernlab` package.

```
model.svm.c1$finalModel
```

```
## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc  (classification)
##  parameter : cost C = 100
##
## Gaussian Radial Basis kernel function.
##  Hyperparameter : sigma =  1
##
## Number of Support Vectors : 4386
##
## Objective Function Value : -3306.48 -728.0289 -516.7598 -1715.672 -660.8427 -485.7019
## Training error : 0.000192
```

3.10.5 Model performance

We test the SVM performance using `test_wine`

```
caret::confusionMatrix(reference = test_wine$quality.c,
                       predict(model.svm.c1,
                               newdata = test_wine[, -c(12, 13, 14, 15)],
                               type = "raw"))
```

```
## Confusion Matrix and Statistics
##
##           Reference
```

```
## Prediction q_5 q_6 q_7 q_3489
##     q_5    398  13   2      3
##     q_6     26 543  14      6
##     q_7      0  10 198      2
##     q_3489   3   1   1     77
##
## Overall Statistics
##
##                Accuracy : 0.9375
##                  95% CI : (0.923, 0.9501)
##     No Information Rate : 0.4372
##     P-Value [Acc > NIR] : < 2e-16
##
##                   Kappa : 0.9061
##
##  Mcnemar's Test P-Value : 0.09136
##
## Statistics by Class:
##
##                      Class: q_5 Class: q_6 Class: q_7 Class: q_3489
## Sensitivity              0.9321     0.9577     0.9209       0.87500
## Specificity              0.9793     0.9370     0.9889       0.99586
## Pos Pred Value           0.9567     0.9219     0.9429       0.93902
## Neg Pred Value           0.9671     0.9661     0.9844       0.99095
## Prevalence               0.3292     0.4372     0.1658       0.06785
## Detection Rate           0.3069     0.4187     0.1527       0.05937
## Detection Prevalence     0.3207     0.4541     0.1619       0.06322
## Balanced Accuracy        0.9557     0.9473     0.9549       0.93543
```

At 66.7%, the model accuracy is comparable to the SVM estimated using `e1071`. Overall, SVM improved the model accuracy significantly over MNL.

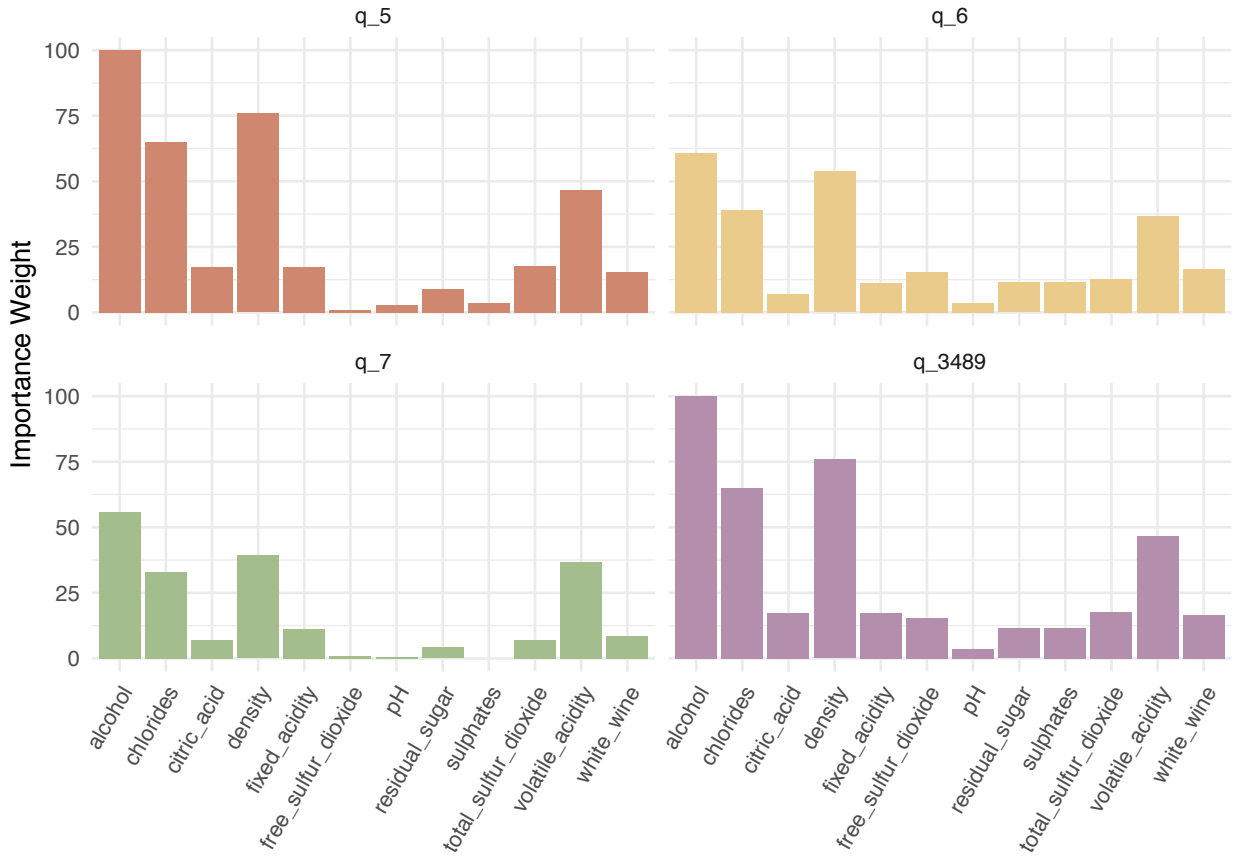
3.10.6 Variable importance

Next, let's plot the variable importance using `ggplot2`.[9]

```
var.imp <- varImp(model.svm.c1,
                  scale = TRUE)

var.imp2 <- var.imp$importance

var.imp2$Chemistry <- rownames(var.imp2)

# Transpose the data
```

[9] `caret` does not report the number of support vectors.

```
var.imp2 <- reshape2::melt(var.imp2, id = c("Chemistry"))

ggplot(data = var.imp2, aes(x = Chemistry, y = value)) +
   geom_col(aes(fill = variable)) +
   facet_wrap(~variable) +
   scale_fill_manual(values = c("#d08770", "#ebcb8b", "#a3be8c", "#b48ead")) +
   xlab(NULL) +
   ylab("Importance Weight") +
   theme_minimal() +
   theme(axis.text.x = element_text(angle = 60, hjust = 1),
            legend.position = "none")
```



Similar to MNL, we have each predictor with different effectiveness in predicting different classes. Clearly `alcohol` content looks like the most important predictor for all the classes.

In summary, SVM performs really well compared to MNL. Note that we have been using `quality.c` as our target variable, which assumes there is no ordering in quality. Thus, even with limited imformation, SVM performed really well.

3.11 Ordinal Regression

In this last section we will use `quality.o` for estimating an ordinal model. Ordinal model can be estimated using several link functions. we will use a logit link.

We will use `ordinal` package and `clm` function.

```
model.ordinal <- ordinal::clm(quality.o ~ .,
                     data = train_wine[,-c(12, 13, 14)])
summary(model.ordinal)
```

```
## formula:
## quality.o ~ fixed_acidity + volatile_acidity + citric_acid + residual_sugar + chlorides + free_sul
## data:    train_wine[, -c(12, 13, 14)]
##
##  link  threshold nobs logLik   AIC     niter max.grad cond.H
##  logit flexible  5200 -4441.32 8910.64 5(0)  7.31e-07 3.7e+02
##
## Coefficients:
##                       Estimate Std. Error z value Pr(>|z|)
## fixed_acidity          0.34702    0.06443   5.386 7.19e-08 ***
## volatile_acidity      -0.67898    0.04604 -14.747  < 2e-16 ***
## citric_acid           -0.05607    0.03595  -1.560 0.118794
## residual_sugar         0.70094    0.09169   7.645 2.10e-14 ***
## chlorides             -0.13134    0.04432  -2.963 0.003044 **
## free_sulfur_dioxide    0.43192    0.04487   9.627  < 2e-16 ***
## total_sulfur_dioxide -0.44547    0.05650  -7.885 3.16e-15 ***
## density               -0.69074    0.14500  -4.764 1.90e-06 ***
## pH                     0.23172    0.04600   5.038 4.71e-07 ***
## sulphates              0.35088    0.03514   9.984  < 2e-16 ***
## alcohol                0.78396    0.07324  10.704  < 2e-16 ***
## white_wine            -0.64337    0.18758  -3.430 0.000604 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Threshold coefficients:
##     Estimate Std. Error z value
## 5|6  -1.2401     0.1459  -8.501
## 6|7   1.4284     0.1461   9.778
```

We have a really nice ordinal model here. Similar to the linear regression model, we have all except `citric_acid` turning up significant at 5% level. The two threshold are also statistically significant indicating that our model identified distinct thresholds to isolate 6 from 5 and 7 from 6. Let's study the model performance using a confusion matrix.

```
confusionMatrix(reference = test_wine$quality.o,
                unlist(predict(model.ordinal,
                               newdata = test_wine[,-c(12, 13, 14, 15)],
                               type = "class")))
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction   5   6   7
##          5 295 140  21
##          6 175 368 168
##          7   6  59  65
##
## Overall Statistics
##
##                Accuracy : 0.5613
##                  95% CI : (0.5338, 0.5885)
##     No Information Rate : 0.4372
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.2828
##
##  Mcnemar's Test P-Value : 6.225e-14
##
## Statistics by Class:
##
##                      Class: 5 Class: 6 Class: 7
## Sensitivity            0.6197   0.6490  0.25591
## Specificity            0.8039   0.5301  0.93768
## Pos Pred Value         0.6469   0.5176  0.50000
## Neg Pred Value         0.7848   0.6604  0.83805
## Prevalence             0.3670   0.4372  0.19584
## Detection Rate         0.2274   0.2837  0.05012
## Detection Prevalence   0.3516   0.5482  0.10023
## Balanced Accuracy      0.7118   0.5896  0.59679
```

As it turns out at 59% the model accuracy is reasonable but not that great. Ordinal regression is the most appropriate model in this case, however. This is because `quality` is actually ordinal.

Exercise: Estimate wine quality model using SVM and `quality.o`. Check whether it has a better model performance than previous models.

In the analysis that I did separately to run SVM using ordinal quality, I found that the out of sample accuracy for SVM was 70%, which is better than any other model we looked at so far.

3.12 Summary

The objective of this chapter was to model one variable, wine quality, treating it as continuous, nominal, and ordinal. Depending on the type of the variable, we ended up using linear model, multinomial logistic regression, support vector machines, and ordinal regression. This exercise suggests that wine quality is not determined by the wine's chemical composition alone.[10]

A common point of failure for MNL, SVM, and ordinal regression was that they incorrectly categorized a lot of quality 7 wines as quality 6 wines. Perhaps this suggests that there is not much difference between these two wines and the models are getting confused. Therefore, model accuracy can be improved if we combine these two groups together.

[10] The original data set stripped out other relevant information such as price and brand of the wine. Perhaps including these variables will lead to a better model performance.

Chapter 4

Car Insurance Calls

In this exercise, we will build a prescriptive model using car insurance sales data. The data set is available on Kaggle from this link: https://www.kaggle.com/kondla/carinsurance

The original objective for this data set as described on Kaggle was this:

We are looking at cold call results. Turns out, same salespeople called existing insurance customers up and tried to sell car insurance. What you have are details about the called customers. Their age, job, marital status, whether the have home insurance, a car loan, etc. As I said, super simple.

What I would love to see is some of you applying some crazy XGBoost classifiers, which we can square off against some logistic regressions. It would be curious to see what comes out on top. Thank you for your time, I hope you enjoy using the data set.

I have changed this *predictive* objective and converted this into a marketing *prescriptive* problem.

4.1 New objectives

1. Build a model that will help marketers segment their customer list
2. Identify a target group for immediate contact and another target group for more expert intervention

Along the way, we will learn many things in modeling such as data preprocessing, hyper parameter tuning, model selection, etc.

4.2 Data

Load the necessary packages. Install the packages that you don't have.

```
library(dplyr)
library(psych)
library(caret)
```

```
library(lubridate)
library(RANN) # Used for KNN
library(lime) # Used for variable importance
library(ggplot2)
library(ROSE) # Used for synthetic samples
library(mice) # Used to impute missing categorical values
```

You can download the csv files from Kaggle or you can directly read the files from my Github repository, which I do next.

We are first going to use only the file names `carInsurance_train.csv`.

```
dt <- read.csv("http://bit.ly/2V8DBkL",
               stringsAsFactors = FALSE)
```

4.2.1 Data exploration

Check out the names of the columns. Compare to the names in the documentation on Kaggle to make sure we are dealing with the same data set. Furthermore, please understand the meaning of each variable.

```
names(dt)
```

```
##  [1] "Id"              "Age"             "Job"
##  [4] "Marital"         "Education"       "Default"
##  [7] "Balance"         "HHInsurance"     "CarLoan"
## [10] "Communication"   "LastContactDay"  "LastContactMonth"
## [13] "NoOfContacts"    "DaysPassed"      "PrevAttempts"
## [16] "Outcome"         "CallStart"       "CallEnd"
## [19] "CarInsurance"
```

Learn the variable classes.

```
sapply(dt, class)
```

```
##              Id             Age             Job         Marital
##       "integer"       "integer"     "character"     "character"
##       Education         Default         Balance     HHInsurance
##     "character"       "integer"       "integer"       "integer"
##         CarLoan   Communication  LastContactDay LastContactMonth
##       "integer"     "character"       "integer"     "character"
##    NoOfContacts      DaysPassed    PrevAttempts         Outcome
##       "integer"       "integer"       "integer"     "character"
##       CallStart         CallEnd    CarInsurance
##     "character"     "character"       "integer"
```

`CallStart` and `CallEnd` are both characters in the data set but in reality they have `hour:minute:second` format. Let's convert them to the right format using `lubridate` package. Separate out hour, minutes, and second from these two variables. Also create a variable`CallDuration` that captures the duration of the call in seconds. Finally, convert all the character variables to factors.

```
dt <- dt %>%
  mutate(CallStart = lubridate::hms(CallStart),
         CallEnd  = lubridate::hms(CallEnd),
         CallStartHour = hour(CallStart),
         CallStartMin = minute(CallStart),
         CallStartSec = second(CallStart),
         CallEndHour = hour(CallEnd),
         CallEndMin = minute(CallEnd),
         CallEndSec = second(CallEnd),
         CallDuration = period_to_seconds(CallEnd) - period_to_seconds(CallStart)) %>%
         ↪
  select(-CallStart, -CallEnd, -Id) %>%
  mutate_if(is.character, as.factor)
```

Get the descriptive

```
psych::describe(dt) %>%
  select(-vars, -trimmed, -mad, -range, -se) %>%
  knitr::kable(digits = 2,
               align = "c",
               caption = "Summary Statistics",
               booktabs = TRUE) # kable prints nice-looking tables.
```

4.2.2 Imputing missing values

Most of the variables have no missing values. However, `Job` and `Education` have a few missing values, which we can easily impute by using a predictive model. `Communication` has 902 missing values and `Outcome` has 3042 missing values. That's too much to just impute. Here we can create a separate category for the missing values.

In case of `Communication`, the missing value is not available. We will create a category for "Not Available". Similarly, for `Outcome` the missing values mean that the person was not contacted before and therefore there is no outcome. Therefore, we will create a category for "None".

Also note that `DaysPassed` takes a value of -1 when the person was not in any previous campaign. Keeping it at -1 is a mistake. Ideally, we should put ∞ in that place. Instead, we will use a very large value. As the maximum days passed is 855, let's use 1,000 as the upper limit.

With that, let's make the changes to our data set.

Table 4.1: Summary Statistics

	n	mean	sd	median	min	max	skew	kurtosis
Age	4000	41.21	11.55	39.0	18	95	0.76	0.49
Job*	3981	5.42	3.21	5.0	1	11	0.20	-1.27
Marital*	4000	2.18	0.63	2.0	1	3	-0.15	-0.56
Education*	3831	2.19	0.67	2.0	1	3	-0.24	-0.80
Default	4000	0.01	0.12	0.0	0	1	8.12	63.95
Balance	4000	1532.94	3511.45	551.5	-3058	98417	9.87	184.73
HHInsurance	4000	0.49	0.50	0.0	0	1	0.03	-2.00
CarLoan	4000	0.13	0.34	0.0	0	1	2.16	2.67
Communication*	3098	1.09	0.28	1.0	1	2	2.95	6.69
LastContactDay	4000	15.72	8.43	16.0	1	31	0.09	-1.07
LastContactMonth*	4000	6.47	3.13	7.0	1	12	-0.36	-1.06
NoOfContacts	4000	2.61	3.06	2.0	1	43	5.24	40.12
DaysPassed	4000	48.71	106.69	-1.0	-1	854	2.53	7.52
PrevAttempts	4000	0.72	2.08	0.0	0	58	8.93	169.74
Outcome*	958	1.88	0.89	2.0	1	3	0.23	-1.69
CarInsurance	4000	0.40	0.49	0.0	0	1	0.40	-1.84
CallStartHour	4000	13.03	2.59	13.0	9	17	-0.04	-1.23
CallStartMin	4000	29.66	17.38	29.0	0	59	-0.01	-1.24
CallStartSec	4000	29.69	17.31	30.0	0	59	-0.04	-1.19
CallEndHour	4000	13.14	2.61	13.0	9	18	-0.03	-1.19
CallEndMin	4000	29.33	17.41	29.0	0	59	0.02	-1.20
CallEndSec	4000	29.10	17.17	29.0	0	59	0.04	-1.19
CallDuration	4000	350.84	342.24	232.0	5	3253	2.23	7.18

```
# Duplicate the original dataset

dt2 <- dt
```

First make the adjustments to `Communication`, `Outcome`, and `DaysPassed`.

```
dt2 <- dt2 %>%
  mutate(Communication = ifelse(is.na(Communication), "Not Available", Communication),
         Outcome = ifelse(is.na(Outcome), "None", Outcome),
         DaysPassed = ifelse(DaysPassed == -1, 1000, DaysPassed))
```

Next impute missing values `Job` and `Education` and use them for missing values. Set a seed for replication in future.

```
set.seed(8934)

miceMod <- mice::mice(subset(dt2, select = -CarInsurance),
                      method = "rf")  # perform mice imputation based on random forest.
```

Generate the completed data.

Table 4.2: Summary Statistics (Post Imputation)

	n	mean	sd	median	min	max	skew	kurtosis
Age	4000	41.21	11.55	39.0	18	95	0.76	0.49
Job*	4000	5.42	3.21	5.0	1	11	0.20	-1.26
Marital*	4000	2.18	0.63	2.0	1	3	-0.15	-0.56
Education*	4000	2.19	0.66	2.0	1	3	-0.24	-0.78
Default	4000	0.01	0.12	0.0	0	1	8.12	63.95
Balance	4000	1532.94	3511.45	551.5	-3058	98417	9.87	184.73
HHInsurance	4000	0.49	0.50	0.0	0	1	0.03	-2.00
CarLoan	4000	0.13	0.34	0.0	0	1	2.16	2.67
Communication*	4000	1.09	0.28	1.0	1	2	2.95	6.69
LastContactDay	4000	15.72	8.43	16.0	1	31	0.09	-1.07
LastContactMonth*	4000	6.47	3.13	7.0	1	12	-0.36	-1.06
NoOfContacts	4000	2.61	3.06	2.0	1	43	5.24	40.12
DaysPassed	4000	809.97	343.85	1000.0	1	1000	-1.31	-0.18
PrevAttempts	4000	0.72	2.08	0.0	0	58	8.93	169.74
Outcome*	4000	1.88	0.89	2.0	1	3	0.23	-1.69
CallStartHour	4000	13.03	2.59	13.0	9	17	-0.04	-1.23
CallStartMin	4000	29.66	17.38	29.0	0	59	-0.01	-1.24
CallStartSec	4000	29.69	17.31	30.0	0	59	-0.04	-1.19
CallEndHour	4000	13.14	2.61	13.0	9	18	-0.03	-1.19
CallEndMin	4000	29.33	17.41	29.0	0	59	0.02	-1.20
CallEndSec	4000	29.10	17.17	29.0	0	59	0.04	-1.19
CallDuration	4000	350.84	342.24	232.0	5	3253	2.23	7.18
CarInsurance	4000	0.40	0.49	0.0	0	1	0.40	-1.84

```
dt3 <- mice::complete(miceMod)
```

Check whether we have all the values imputed now.

```
anyNA(dt3)
```

```
## [1] FALSE
```

Now let's look at the summary.

```
dt3$CarInsurance <- dt2$CarInsurance

psych::describe(dt3) %>%
  select(-vars, -trimmed, -mad, -range, -se) %>%
  knitr::kable(digits = 2,
               align = "c",
               caption = "Summary Statistics\n(Post Imputation)",
               booktabs = TRUE)
```

Finally, we will create dummy variables for all the factors. For this, we will use `dummyVars` function from `caret` package. It will drop the dependent variable, `CarInsurance` from the data, so we will first save it into a vector and then add it back.

Also, `predict` function will create a `matrix`, which we need to convert into a `data frame`.

```
dt4 <- predict(dummyVars(CarInsurance ~ .,
                         data = dt3,
                         fullRank = TRUE # Drops the reference category
                         ),
               newdata = dt3) %>%
  data.frame()

dt4$CarInsurance <- dt2$CarInsurance

# Convert CarInsurance to a factor
dt4 <- dt4 %>%
  mutate(CarInsurance = as.factor(ifelse(CarInsurance == 0, "No", "Yes")))
```

4.2.3 Training and test set

Before we build our predictive model, we will split the sample into a training set and a test set. Usually I use 80 - 20 split. As our sample size is pretty large, this split is reasonable. We will build our model using training data and then evaluate its out-of-sample performance on the test data.

For this we will use `caret`'s `createDataPartition()` function. This function takes a factor as input along with the percentage of split. In our case, we want to split the data in training and testing sets but we want to make sure that the proportion of Yes and No for `CarInsurance` remains the same in the two splits. The output of this function is a numeric vector with values corresponding to the row numbers that we want to keep in the training set. Obviously, the test set has all the rows which are discarded by `createDataPartition()`.

```
# Create a vector with row numbers corresponding to the training data

index <- caret::createDataPartition(dt4$CarInsurance,
                                    p = 0.8,
                                    list = FALSE) # caret returns a list by default.

dt4_train <- dt4[index, ]
dt4_test <- dt4[-index, ]
```

4.3 Building the predictive model

We will use random forest for building the predictive model. First define the training controls. The only meaningful hyper parameter that makes substantial difference in accuracy is `mtry`

which is the number of variables to use for building each tree in the random forest. You can tune this hyper parameter by using grid search.

```
trControl <- trainControl(method = "cv", #crossvalidation
                          number = 10,   # 10 folds
                          search = "grid",
                          classProbs = TRUE  #computes class probabilities
                          )

tuneGrid_large <- expand.grid(mtry = c(1:(ncol(dt4) - 2)))
```

Now train the model. The code in the next block is for demonstration purposes and I advice you not to run it during the class. This is because it will take several minutes if not hours to execute.

We derive the model by using `train()` function form `caret`. We first specify the formula, which in our case is `CarInsurance` as a function of all the variable sin the model. Next, we specify the data set to be used. `caret` has numerous machine learning and statistical methods (258 in all). For random forest, we will use `rf` method. With this method, under the hood, `caret` is using `randomForest` package. But note that other alternatives for random forest such as `ranger` are available as well.[1]

For classification, we will use "Accuracy" as the metric to maximize. Then we provide the tuning grid and training control objects, and finally select the number of trees.[2]

Warning: This will take several minutes or even hours to run!

```
set.seed(9933)

modelRF_large <- train(CarInsurance ~ . ,
                  data = dt4_train,
                  method = "rf",
                  metric = "Accuracy",
                  tuneGrid = tuneGrid_large,
                  trControl = trControl,
                  ntree = 1000)
```

Now print the model to take a look at the accuracies.

```
print(modelRF_large)
```

```
## Random Forest
##
```

[1] How about you trying out other random forest methods? Check out https://topepo.github.io/caret/train-models-by-tag.html#Random_Forest

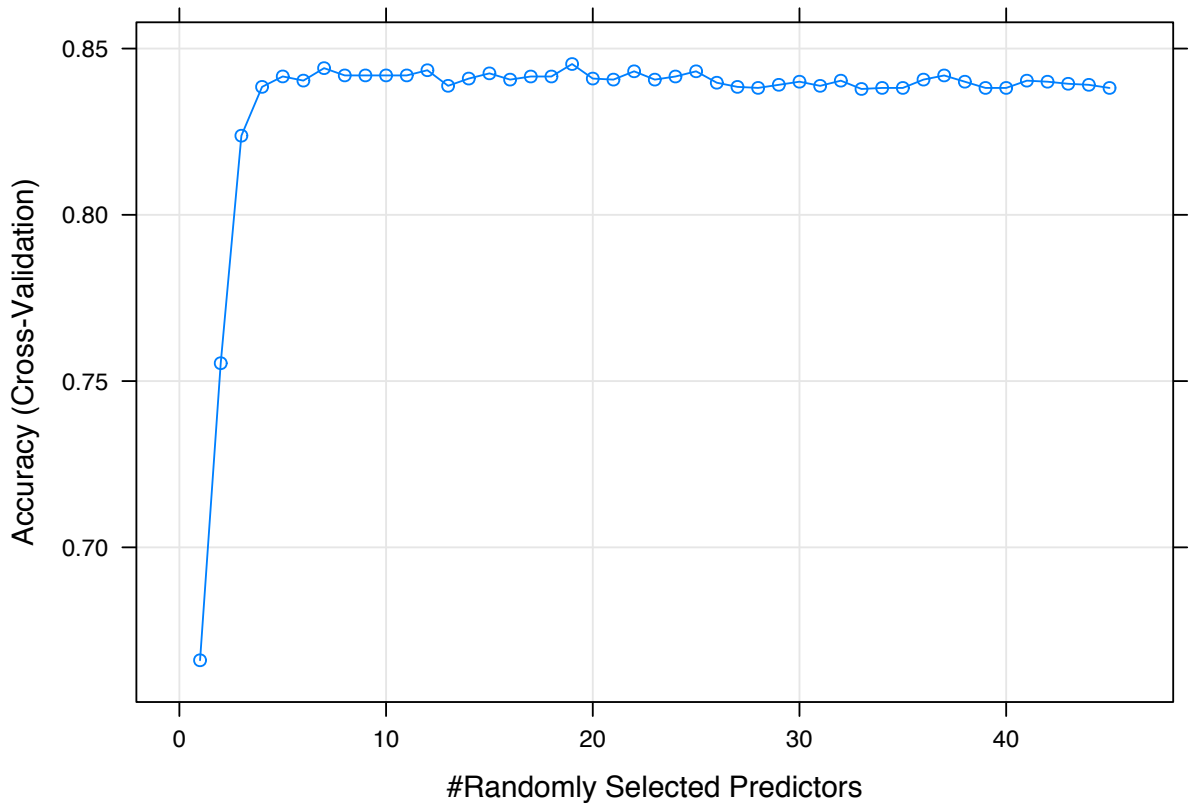
[2] Note that `caret` does not treat the number of trees as a hyper parameter. Therefore, you can't use it in the tuning grid. If, however, you are really interested in tweaking the number of trees, you should use a for loop.

```
## 3201 samples
##   46 predictor
##    2 classes: 'No', 'Yes'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 2882, 2880, 2880, 2881, 2881, 2881, ...
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa
##    1    0.6660384  0.2024168
##    2    0.7553787  0.4543552
##    3    0.8237960  0.6271341
##    4    0.8384777  0.6632234
##    5    0.8416154  0.6720805
##    6    0.8403644  0.6693719
##    7    0.8441106  0.6775140
##    8    0.8419221  0.6734917
##    9    0.8419211  0.6739009
##   10    0.8419230  0.6739344
##   11    0.8419221  0.6738316
##   12    0.8434836  0.6774717
##   13    0.8387951  0.6673813
##   14    0.8409777  0.6717585
##   15    0.8425412  0.6750726
##   16    0.8406633  0.6718024
##   17    0.8415969  0.6734528
##   18    0.8415998  0.6731690
##   19    0.8453489  0.6815814
##   20    0.8409767  0.6721318
##   21    0.8406662  0.6714256
##   22    0.8431594  0.6767590
##   23    0.8406623  0.6713721
##   24    0.8415978  0.6734469
##   25    0.8431604  0.6770513
##   26    0.8397297  0.6698172
##   27    0.8384738  0.6671387
##   28    0.8381691  0.6663353
##   29    0.8390988  0.6685000
##   30    0.8400353  0.6704002
##   31    0.8387863  0.6678400
##   32    0.8403469  0.6711783
##   33    0.8378488  0.6660378
##   34    0.8381642  0.6662424
##   35    0.8381603  0.6664330
##   36    0.8406643  0.6717384
##   37    0.8419133  0.6744978
```

```
##    38    0.8400324  0.6705347
##    39    0.8381603  0.6666285
##    40    0.8381623  0.6666150
##    41    0.8403478  0.6707452
##    42    0.8400314  0.6705115
##    43    0.8394084  0.6692103
##    44    0.8390930  0.6687031
##    45    0.8381584  0.6666179
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 19.
```

We can also make a plot for an easy comparison.

```
plot(modelRF_large)
```



We are getting the highest accuracy when `mtry = 19` so I am going to set `mtry` to 19 for this example. We will execute the code below in the class. However, if you are curious, you could use `mtry = 7` as well given that there is a minor difference between the two accuracies. Furthermore, a smaller number of trees is preferred over a larger number because it is likely to perform better out of sample.

4.3.1 Random forest with fixed `mtry`

Run this code in the class instead of the grid search above.

```
trControl <- trainControl(method = "cv",
                          number = 10,
                          search = "grid",
                          classProbs = TRUE)

tuneGrid <- expand.grid(mtry = 19)
```

Next, train the model using the above training controls.

```
set.seed(9999)

modelRF <- train(CarInsurance ~ . ,
                 data = dt4_train,
                 method = "rf",
                 metric = "Accuracy",
                 tuneGrid = tuneGrid,
                 trControl = trControl,
                 ntree = 1000)
```

```
print(modelRF)
```

```
## Random Forest
##
## 3201 samples
##   46 predictor
##    2 classes: 'No', 'Yes'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 2880, 2881, 2881, 2881, 2881, 2880, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.8422473  0.6749728
##
## Tuning parameter 'mtry' was held constant at a value of 19
```

Our model has a decent resampling accuracy of 84.2%. Kappa is 0.67, which is also fairly acceptable. [3]

[3] In practice Kappa > 0.75 suggests very good model accuracy.

```
varImp(modelRF, scale = TRUE)
```

```
## rf variable importance
##
##   only 20 most important variables shown (out of 46)
##
##                             Overall
## CallDuration               100.000
## Age                         16.384
## Outcome3                    15.770
## Balance                     13.517
## CallEndSec                  10.594
## LastContactDay              10.296
## CallEndMin                   9.574
## CallStartSec                 9.527
## CallStartMin                 9.335
## DaysPassed                   8.947
## CommunicationNot.Available   8.469
## HHInsurance                  8.132
## CallEndHour                  5.118
## NoOfContacts                 4.962
## CallStartHour                4.742
## PrevAttempts                 3.553
## LastContactMonth.mar         3.524
## LastContactMonth.aug         2.892
## CarLoan                      2.026
## LastContactMonth.jun         2.009
```

Variance importance suggests that `CallDuration` is the single-most important variable! Let's talk more about this below.

```
confusionMatrix(predict(modelRF, select(dt4_test, -CarInsurance)),
                reference = dt4_test$CarInsurance,
                positive = "Yes")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  No Yes
##        No  461  12
##        Yes  18 308
##
##                Accuracy : 0.9625
##                  95% CI : (0.9468, 0.9745)
##     No Information Rate : 0.5995
```

```
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.9221
##
##  Mcnemar's Test P-Value : 0.3613
##
##             Sensitivity : 0.9625
##             Specificity : 0.9624
##          Pos Pred Value : 0.9448
##          Neg Pred Value : 0.9746
##              Prevalence : 0.4005
##          Detection Rate : 0.3855
##    Detection Prevalence : 0.4080
##       Balanced Accuracy : 0.9625
##
##        'Positive' Class : Yes
##
```

The confusion matrix suggests that our model performs well outside the sample as well. However, the variable importance calculated above suggests that `CallDuration` may have a spurious relationship between the likelihood to buy insurance. This is because when a person is interested in buying the insurance, he/she will spend more time on the call.

For a purely predictive task, this is not a concern. If, for example, given all the information in the data set, we want to predict whether a person bought insurance or not, we will do well with the model we built. However, consider this problem from a marketing manager's perspective. The manager wants to know whether it makes sense to even make a call to a customer. Because the real cost here is the cost of contacting a prospective buyer. So in order to reduce the cost of contacting them, they would like to build a model based on the information that does not include calls.

So, `CallDuration` might be a good metric for predicting insurance purchase but it is not a good metric for prescribing who to call. This is because, 1) the call has not happened yet and 2) one can't simply increase the call length and expect the prospect to buy insurance. If call length is the metric to optimize, salespeople will likely game the system and talk nonsense on the phone just to extend the call.

4.3.2 Remove call-related variables

Let's rerun the example after removing the call related variables.

```
set.seed(9999)

modelRF2 <- train(CarInsurance ~ . ,
                  data = select(dt4_train, -starts_with("Call")),
                  method = "rf",
                  metric = "Accuracy",
                  tuneGrid = tuneGrid,
```

```
                trControl = trControl,
                ntree = 1000)
```

Print the model

```
print(modelRF2)
```

```
## Random Forest
##
## 3201 samples
##   39 predictor
##    2 classes: 'No', 'Yes'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 2880, 2881, 2881, 2881, 2881, 2880, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.7400749  0.4361087
##
## Tuning parameter 'mtry' was held constant at a value of 19
```

Now, the model accuracy went down significantly. Let's see which variables are important.

```
varImp(modelRF2, scale = TRUE)
```

```
## rf variable importance
##
##   only 20 most important variables shown (out of 39)
##
##                             Overall
## Balance                     100.000
## Age                          84.169
## LastContactDay               69.435
## Outcome3                     36.007
## NoOfContacts                 34.916
## DaysPassed                   27.374
## CommunicationNot.Available   21.333
## HHInsurance                  15.995
## PrevAttempts                 11.600
## Marital.married               9.927
## Education.secondary           9.406
## CarLoan                       9.376
## LastContactMonth.aug          8.125
```

```
## Job.technician               8.075
## Education.tertiary           7.604
## LastContactMonth.jun         7.493
## LastContactMonth.jul         7.460
## Job.management               7.333
## LastContactMonth.may         7.120
## Job.blue.collar              7.070
```

`Balance`, `Age`, and `LastContactDay` are the three most important variables predicting the likelihood to buy insurance. We do not know the directions of their effects. We will learn more about that in a moment.

Let's see what the new confusion matrix shows us.

```
confusionMatrix(predict(modelRF2, select(dt4_test, -CarInsurance,
↪  -starts_with("Call"))),
                reference = dt4_test$CarInsurance,
                positive = "Yes")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  No Yes
##        No  470  29
##        Yes   9 291
##
##                Accuracy : 0.9524
##                  95% CI : (0.9353, 0.9661)
##     No Information Rate : 0.5995
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.8999
##
##  Mcnemar's Test P-Value : 0.002055
##
##             Sensitivity : 0.9094
##             Specificity : 0.9812
##          Pos Pred Value : 0.9700
##          Neg Pred Value : 0.9419
##              Prevalence : 0.4005
##          Detection Rate : 0.3642
##    Detection Prevalence : 0.3755
##       Balanced Accuracy : 0.9453
##
##        'Positive' Class : Yes
##
```

Our model performs really poorly out of sample. In particular, we misclassified about 51% (156 / 320) potential purchases. Note that if the marginal cost of a call is much lower compared to the marginal cost of losing a customer, our model is performing really poorly with only 51% sensitivity.

4.4 Changing probability cutoff

Note that `caret` is using a probability cutoff of 0.5 to determine whether a person will buy insurance or not. We can change that cutoff to 0.3 to see whether we get better results.

```
predict_custom <- predict(modelRF2,
                          select(dt4_test, -CarInsurance, -starts_with("Call")),
                          type = "prob") %>%
  mutate(new_class = factor(ifelse(Yes >= 0.3, "Yes", "No"))) %>%
  select(new_class)


confusionMatrix(predict_custom$new_class,
                reference = dt4_test$CarInsurance,
                positive = "Yes")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  No Yes
##        No  430  13
##        Yes  49 307
##
##                Accuracy : 0.9224
##                  95% CI : (0.9016, 0.94)
##     No Information Rate : 0.5995
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.8414
##
##  Mcnemar's Test P-Value : 8.789e-06
##
##             Sensitivity : 0.9594
##             Specificity : 0.8977
##          Pos Pred Value : 0.8624
##          Neg Pred Value : 0.9707
##              Prevalence : 0.4005
##          Detection Rate : 0.3842
##    Detection Prevalence : 0.4456
##       Balanced Accuracy : 0.9285
```

```
##
##        'Positive' Class : Yes
##
```

With a revised cutoff of 0.3, although we now identify too many prospective buyers, we do not unnecessarily leave out a lot of prospective customers. This is also a good lesson for us. We can't improve the overall accuracy of the model just by changing the default cutoff of 0.5.

4.5 Aside: Variable importance using `lime` (optional)

You may skip this part and move down to Section 4.6

In prescriptive analytics we are also interested in knowing the direction of the effect. For example, we expect that `Balance` is increasing the probability that a person buys insurance. However, `caret` does not tell us whether this is indeed true. In order to tackle this issue, we will use `lime` package. [4] I won't be able to go into the details here.

Note that we are using only 20 observations from the `dt4_test` data set to save time.

```
explainer_rf2 <- lime::lime(select(dt4_train, -starts_with("Call")),
                            modelRF2,
                            n_bins = 5) # number of bins for continuous variables

explanation_rf2 <- lime::explain(
  x = select(dt4_test[1:20, ], -CarInsurance, -starts_with("Call")),
  explainer = explainer_rf2,
  n_permutations = 5000, # default
  dist_fun = "gower",    # default
  kernel_width = 0.75,   # default
  n_features = 10,
  feature_select = "highest_weights",
  labels = "Yes"         # the label for the event "buy"
)
```

Plot the features

```
plot_explanations(explanation_rf2)
```

[4] LIME stands for Local Interpretable Model-agnostic Explanations

`lime` shows the results for each case separately. For instance, out of the 20 observations, 16 show increased probability of purchase if `NoOfContacts` is <= 2. Fore more details, please visit https://uc-r.github.io/lime.

4.6 Tweaking the model (and data!)

We have several tools at our disposal to improve the model performance. My first advice is to use grid search and tune `mtry`. If you have time or a powerful computer at your disposal, tune the number of trees as well. We have fixed it at 1,000.

4.6.1 Dropping irrelevant variables

We can drop a few less important variables from our model as they might be adding noise. Let's keep only the variables with scaled importance more than 10

```
impvar <- varImp(modelRF2, scale = TRUE)[[1]] %>%
  tibble::rownames_to_column() %>%
  filter(Overall > 10) %>%
  pull(rowname)
```

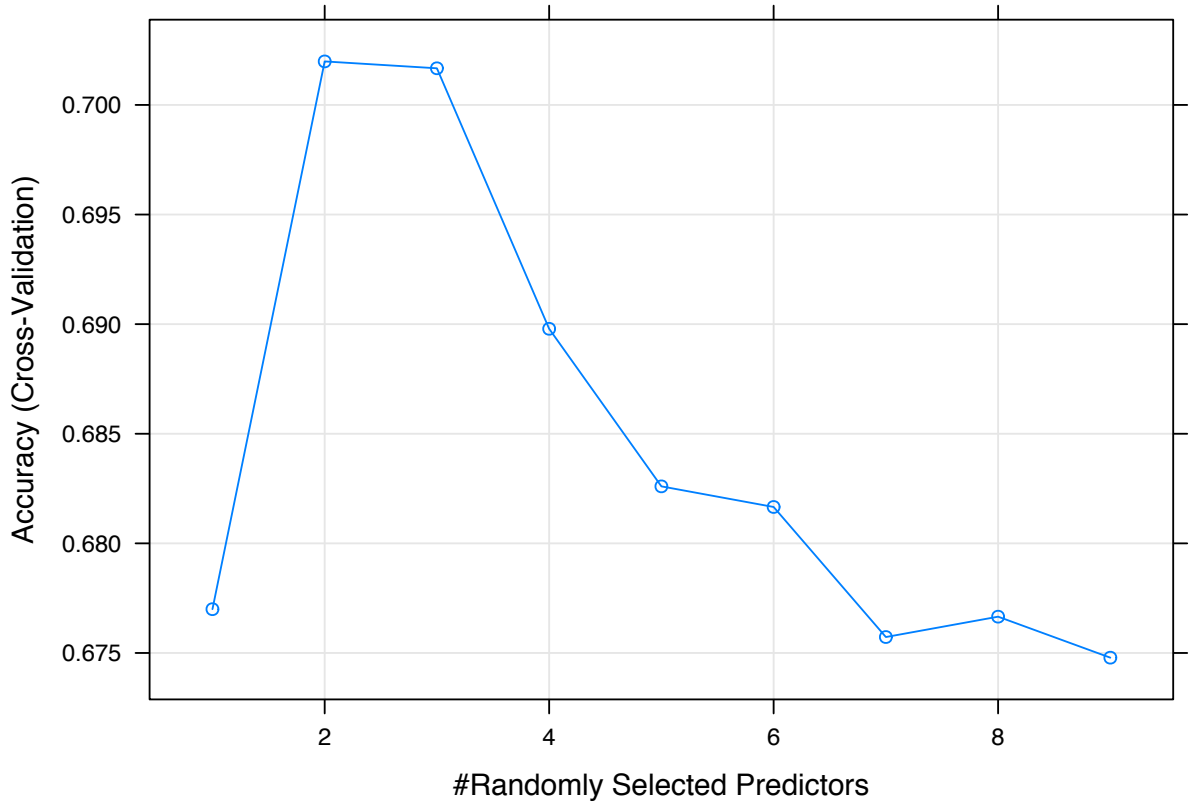
Now, `impvar` is a vector with the important variables. The next part will take some time to finish running because I am going to try multiple values of `mtry`. However, as the number of variables is small, this will be much quicker than the larger model above.

```
set.seed(9999)

modelRF3 <- train(CarInsurance ~ . ,
                  data = select(dt4_train, CarInsurance, impvar),
                  method = "rf",
                  metric = "Accuracy",
                  tuneGrid = expand.grid(mtry = c(1:9)),
                  trControl = trControl,
                  ntree = 1000)
```

```
print(modelRF3)
```

```
## Random Forest
##
## 3201 samples
##    9 predictor
##    2 classes: 'No', 'Yes'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 2880, 2881, 2881, 2881, 2881, 2880, ...
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa
##   1     0.6769974  0.2468218
##   2     0.7019848  0.3267983
##   3     0.7016714  0.3375243
##   4     0.6897875  0.3214448
##   5     0.6826000  0.3125865
##   6     0.6816596  0.3136671
##   7     0.6757269  0.3008956
##   8     0.6766556  0.3027233
##   9     0.6747855  0.2987287
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 2.
```

```
plot(modelRF3)
```



Clearly, the solution with `mtry` = 2 is the best in this scenario. Let's see how the model performs out of the sample on the test set.

```
confusionMatrix(predict(modelRF3, select(dt4_test, impvar)),
                reference = dt4_test$CarInsurance,
                positive = "Yes")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  No Yes
##        No  467 173
##        Yes  12 147
##
##                Accuracy : 0.7685
##                  95% CI : (0.7376, 0.7973)
##     No Information Rate : 0.5995
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.4739
##
##  Mcnemar's Test P-Value : < 2.2e-16
```

```
##
##             Sensitivity : 0.4594
##             Specificity : 0.9749
##          Pos Pred Value : 0.9245
##          Neg Pred Value : 0.7297
##              Prevalence : 0.4005
##          Detection Rate : 0.1840
##    Detection Prevalence : 0.1990
##       Balanced Accuracy : 0.7172
##
##        'Positive' Class : Yes
##
```

By dropping the variables we did not increase the accuracy of the model. In fact, the sensitivity is now lower. Therefore, we will not use this model further.

4.6.2 Balancing classes

Note that the proportion of "Yes" and "No" in our model is not 50:50.

```
table(dt4_train$CarInsurance)
```

```
##
##   No  Yes
## 1917 1284
```

We can balance these classes and hope to improve classification accuracy. For this we will use `ROSE` function from `ROSE` package.[5]

`ROSE` function creates synthetic samples in order to balance the classes. Below, I keep the sample size the same, so in order to balance the two classes, `ROSE` will undersample from "No" and oversample from "Yes". As `ROSE()` returns a list, we retain the data frame that's relevant for us. Also note that we can specify a random number seed in the function.

```
dt4_train2 <- ROSE::ROSE(CarInsurance ~ .,
                  data = select(dt4_train, -starts_with("Call")),
                  N = 3201,
                  p = 0.5,
                  seed = 305)$data
```

Check the class balance.

[5] Read more about this here: https://journal.r-project.org/archive/2014-1/menardi-lunardon-torelli.pdf

```
table(dt4_train2$CarInsurance)
```

```
##
##   No  Yes
## 1590 1611
```

Now the two classes are almost equally balanced. Let's use the new synthetic sample.

```
set.seed(9999)

modelRF4 <- train(CarInsurance ~ . ,
                  data = dt4_train2,
                  method = "rf",
                  metric = "Accuracy",
                  tuneGrid = tuneGrid,
                  trControl = trControl,
                  ntree = 1000)
```

```
print(modelRF4)
```

```
## Random Forest
##
## 3201 samples
##   39 predictor
##    2 classes: 'No', 'Yes'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 2881, 2881, 2880, 2881, 2881, 2881, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.9222099  0.8443877
##
## Tuning parameter 'mtry' was held constant at a value of 19
```

Wow, look at that! By using synthetic sampling, we increased the accuracy of our model to 92.2%. But does that also help us improve the out-of-sample performance?

```
confusionMatrix(predict(modelRF4, select(dt4_test, -CarInsurance,
↪  -starts_with("Call"))),
                reference = dt4_test$CarInsurance,
                positive = "Yes")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  No Yes
##        No  467 231
##        Yes  12  89
##
##                Accuracy : 0.6959
##                  95% CI : (0.6627, 0.7276)
##     No Information Rate : 0.5995
##     P-Value [Acc > NIR] : 9.818e-09
##
##                   Kappa : 0.2855
##
##  Mcnemar's Test P-Value : < 2.2e-16
##
##             Sensitivity : 0.2781
##             Specificity : 0.9749
##          Pos Pred Value : 0.8812
##          Neg Pred Value : 0.6691
##              Prevalence : 0.4005
##          Detection Rate : 0.1114
##    Detection Prevalence : 0.1264
##       Balanced Accuracy : 0.6265
##
##        'Positive' Class : Yes
##
```

Looks like our new model has worse out-of-sample performance. :(

This situation where you have an improved in-sample performance but a worse out-of-sample performance is known as overfitting.

Due to the poor predictive power of the model built on synthetic sample, let's go back to our original model, `modelRF2` for the rest of the analysis.

4.7 Making prescriptions

In this section, we will use our predictive model to get probabilities for a set of potential customers. we will then use those probabilities to create segments to target.

First, get the list of prospects.

```
prospects <- read.csv("http://bit.ly/2VSzAVT",
                      stringsAsFactors = FALSE)
```

We will have to perform the same adjustments that we did on the training data.

```
prospects2 <- prospects %>%
  mutate(CallStart = lubridate::hms(CallStart),
         CallEnd  = lubridate::hms(CallEnd),
         CallStartHour = hour(CallStart),
         CallStartMin = minute(CallStart),
         CallStartSec = second(CallStart),
         CallEndHour = hour(CallEnd),
         CallEndMin = minute(CallEnd),
         CallEndSec = second(CallEnd),
         CallDuration = period_to_seconds(CallEnd) - period_to_seconds(CallStart)) %>%
         ↪
  select(-CallStart, -CallEnd) %>%
  mutate_if(is.character, as.factor) %>%
  mutate(Communication = ifelse(is.na(Communication),
                                "Not Available",
                                Communication),
         Outcome = ifelse(is.na(Outcome), "None", Outcome),
         DaysPassed = ifelse(DaysPassed == -1, 1000, DaysPassed))
```

Impute missing values as before.

```
set.seed(8934)

miceMod2 <- mice::mice(subset(prospects2, select = -CarInsurance),
                       method = "rf")
```

Generate the completed data, create dummy variables, and convert `CarInsurance` to a factor.

```
prospects3 <- mice::complete(miceMod2)


prospects3$CarInsurance <- prospects2$CarInsurance

prospects4 <- predict(dummyVars(CarInsurance ~ .,
                          data = prospects3,
                          fullRank = TRUE),
                      newdata = prospects3) %>%
  data.frame()

prospects4$CarInsurance <- prospects2$CarInsurance

prospects4 <- prospects4 %>%
  mutate(CarInsurance = as.factor(ifelse(CarInsurance == 0, "No", "Yes")))
```

4.8 Probabilities

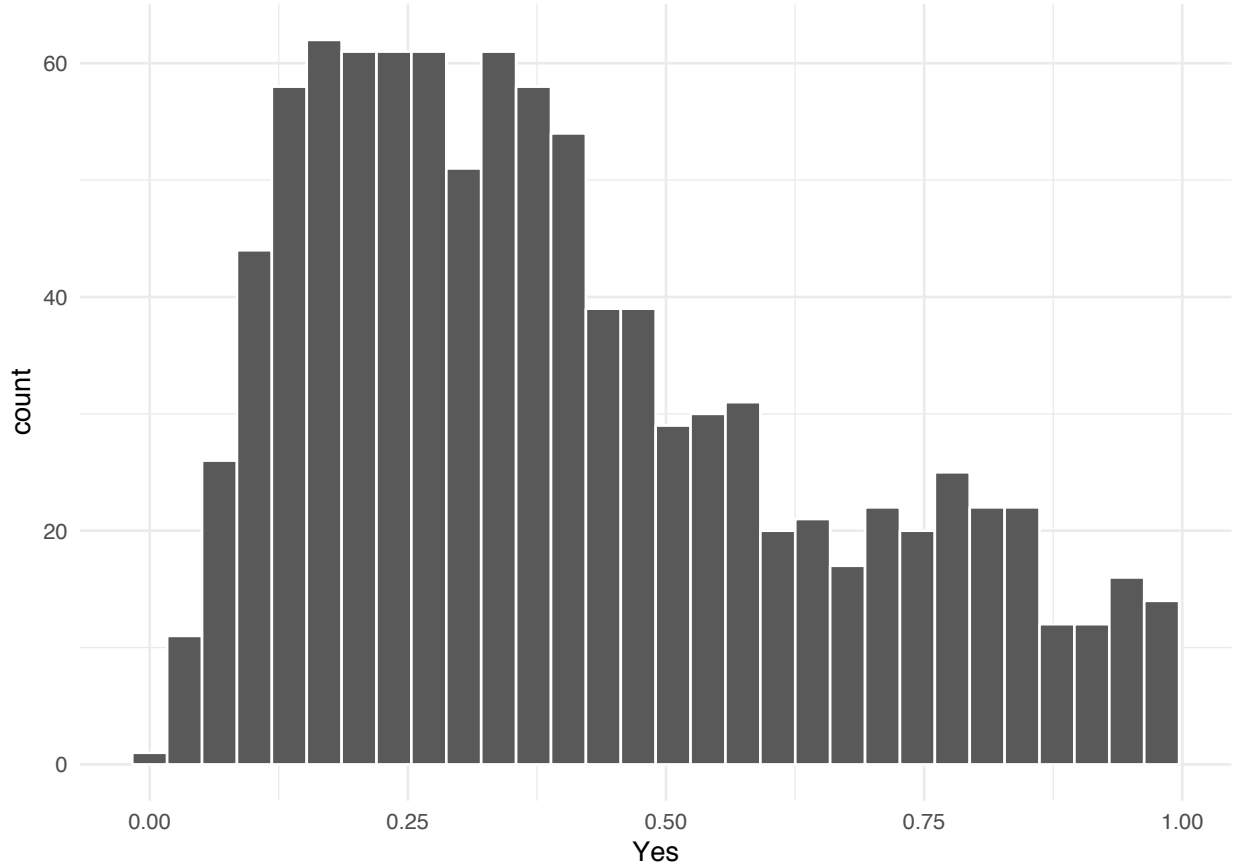
Predict probabilities of insurance purchase using `modelRF2`.

```
pred_prob <- predict(modelRF2,
                     select(prospects4, -Id, -starts_with("Call")),
                     type = "prob")
```

Plot these probabilities

```
ggplot(pred_prob, aes(x = Yes)) +
  geom_histogram(color = "white") +
  theme_minimal()
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```



Looks like the histogram is positively skewed, which suggests that many people on the list are unlikely to buy. However, marketers don't have to stick to the 50% probability cutoff. Instead, we can prescribe cutoffs based on our own experience in the past. For instances, we can use a rule to create segments for targeting. Here I give you a few examples:

1. Call the prospects with at least 70% probability right away.
2. For the prospects with probabilities between 20% and 70%, use expert salespersons to call.
3. Ignore all the prospects with less than 20% probability.

Clearly, these cutoffs seem arbitrary. In reality, such decisions are made after some brainstorming and doing some experimentation.

This is the end of the exercise. [6]

[6] What other prescriptions can you give to the company?

Chapter 5

Twitter Sentiment

Social media analysis is highly popular. This chapter has four exercises using Twitter data. You can extend it to other social networks too. We will access Twitter using their official application programming interface (API). However, Facebook, Instagram, and LinkedIn don't have easily accessible APIs any more so our choices are limited.

5.1 Tasks to complete

Four objectives of this exercise are as follows —

1. Collect a large number of tweets for a trending hashtag
2. Create a map of the tweets based on their location
3. Do sentiment analysis on the tweets
4. Create a wordcloud

5.2 Twitter API access

In order to get access to Twitter API, you will have to request for a developer account. It's a free account but Twitter controls the approval process and it is not automatic. It can take anywhere from a few minutes to a few days. However, your chances of getting a quick approval are higher if you use your `.edu` email address to create an account. Also note that you will have to provide a valid mobile number. We will use `rtweet` package written by Mike Kearney, who has written a nice vignette for getting Twitter authentication. Please follow all the steps and create an authentication token first. The vignette is available here: https://rtweet.info/articles/auth.html

I strongly recommend that you use the second method titled **Access token/secret method** in this vignette. For this exercise, I assume that you have this token generated and stored on your hard drive. When you save it on your hard drive, **please don't use any file extension.**

Note that without the token you will not be able to obtain data from Twitter. However, for the sake of this exercise, I have already downloaded Twitter data and made available to you from this link: http://bit.ly/2DVbGia. It's an RDS file so once you download it on your computer, read it using `readRDS()` function.[1]

5.3 Collect tweets

The first order of the business is to collect the tweets. Ideally, you would like to collect the tweets about a topic that is trending on the day you are reading this text. Currently[2] for me it is "Liverpool" referring to the football (soccer) team in the UK. Liverpool beat Barcelona to qualify for the Champions League final.

Load up the necessary libraries

```
library(rtweet)  # Twitter package
library(dplyr)
library(ggplot2)
library(sf)   # For making maps
library(usmap)
library(reshape2)

# Packages for text analysis and wordcloud
library(tm)
library(syuzhet)
library(tidytext)
library(ggwordcloud)
library(tm)
library(SnowballC)
library(wordcloud)
library(RColorBrewer)
```

Next, load your Twitter token into the environment. Note that you are going to read this file from the place you saved your token. I am assuming that your token is saved with "twitter_token" as the name. Again, note that there is no file extension because you saved it without an extension.

```
load(here::here("twitter_token"))
```

After this, you should see your Twitter token in the Global Environment on the top right window in RStudio.

[1] You have to do this only if you do not have your own Twitter credentials.
[2] May 7, 2019

5.3.1 Get the tweets mentioning "Liverpool"

We will use `search_tweets()` function to search and download tweets. Twitter's rate limit restricts downloading 18,000 tweets every 15 minutes. If you want to download more tweets, you will have to accordingly wait.[3] We will download 18,000 tweets from the USA sent out in English. You can easily change these as you want.

```
lp <-  search_tweets(q = "Liverpool",
                     lang = 'en',
                     geocode = lookup_coords("usa"),
                     n = 18000,
                     include_rts = FALSE, # exclude retweets
                     )
```

The search and download usually takes only about a couple of minute or so. Twitter's API returns all the data in `json` format but `rtweet` converts it into a `data.frame`.

5.4 Data exploration

Twitter gives out a lot of information but `rtweet` returns only some of it. Yet, `lp` is a large data frame with 17,905 observations and 88 variables. Let's see what variable names we have.

```
names(lp)
```

```
##  [1] "user_id"                "status_id"
##  [3] "created_at"             "screen_name"
##  [5] "text"                   "source"
##  [7] "display_text_width"     "reply_to_status_id"
##  [9] "reply_to_user_id"       "reply_to_screen_name"
## [11] "is_quote"               "is_retweet"
## [13] "favorite_count"         "retweet_count"
## [15] "hashtags"               "symbols"
## [17] "urls_url"               "urls_t.co"
## [19] "urls_expanded_url"      "media_url"
## [21] "media_t.co"             "media_expanded_url"
## [23] "media_type"             "ext_media_url"
## [25] "ext_media_t.co"         "ext_media_expanded_url"
## [27] "ext_media_type"         "mentions_user_id"
## [29] "mentions_screen_name"   "lang"
## [31] "quoted_status_id"       "quoted_text"
## [33] "quoted_created_at"      "quoted_source"
```

[3] For instance, if you want to download 20,000 tweets, Twitter will first download 18,000 tweets and then the rate limit will set in. You will have to wait for about 15 minutes after which the remaining 2,000 tweets will be downloaded.

```
## [35] "quoted_favorite_count"   "quoted_retweet_count"
## [37] "quoted_user_id"          "quoted_screen_name"
## [39] "quoted_name"             "quoted_followers_count"
## [41] "quoted_friends_count"    "quoted_statuses_count"
## [43] "quoted_location"         "quoted_description"
## [45] "quoted_verified"         "retweet_status_id"
## [47] "retweet_text"            "retweet_created_at"
## [49] "retweet_source"          "retweet_favorite_count"
## [51] "retweet_retweet_count"   "retweet_user_id"
## [53] "retweet_screen_name"     "retweet_name"
## [55] "retweet_followers_count" "retweet_friends_count"
## [57] "retweet_statuses_count"  "retweet_location"
## [59] "retweet_description"     "retweet_verified"
## [61] "place_url"               "place_name"
## [63] "place_full_name"         "place_type"
## [65] "country"                 "country_code"
## [67] "geo_coords"              "coords_coords"
## [69] "bbox_coords"             "status_url"
## [71] "name"                    "location"
## [73] "description"             "url"
## [75] "protected"               "followers_count"
## [77] "friends_count"           "listed_count"
## [79] "statuses_count"          "favourites_count"
## [81] "account_created_at"      "verified"
## [83] "profile_url"             "profile_expanded_url"
## [85] "account_lang"            "profile_banner_url"
## [87] "profile_background_url"  "profile_image_url"
```

It is not possible to provide description for each of these variables. However, the variables are a mix of user data and tweet data. For instance, `user_id` tells us the unique user id while `status_id` is the unique id given to this tweet.

I would like to show you two interesting variables: `source` and `verified`. The first one contains the information on the device that was used to send out the tweet. The second variable tells us whether the person has a verified Twitter account.

Using `count()` function from `dplyr` we can see which device is the most popular. As we may have the same person tweeting multiple times, we will keep only distinct `user_id-source` pairs.

```
lp %>%
  distinct(user_id, source) %>%
  count(source, sort = TRUE) %>%
  top_n(10)
```

```
## Selecting by n

## # A tibble: 10 x 2
##    source                  n
```

```
##    <chr>               <int>
##  1 Twitter for iPhone   8810
##  2 Twitter for Android  3485
##  3 Twitter Web Client   1251
##  4 Twitter Web App       655
##  5 Facebook              214
##  6 TweetDeck             203
##  7 Twitter for iPad      170
##  8 Tweetbot for iOS       60
##  9 Instagram              55
## 10 IFTTT                  47
```

This particular pattern is more prevalent in the US. In the other countries, Twitter for Android usually tops the list.

How many verified accounts do we have in our sample?

```
lp %>%
  distinct(user_id, verified) %>%
  count(verified, sort = TRUE)
```

```
## # A tibble: 2 x 2
##   verified     n
##   <lgl>    <int>
## 1 FALSE    14695
## 2 TRUE       370
```

It's impressive that we have 370 verified accounts. Later we will see whether their twitter sentiment is different from non-verified accounts.

5.5 Mapping tweets

As the data obtained is from the US, we should be able to make a map of these tweets very easily. Most people on Twitter do not disclose their location. But as we have a lot of tweets, we will find some of them with their location public. Twitter returned `geo_coords` variable, which has the latitude and longitude. We will use a handy function from `rtweet` to extract those.

```
lp_geo <- lat_lng(lp)
```

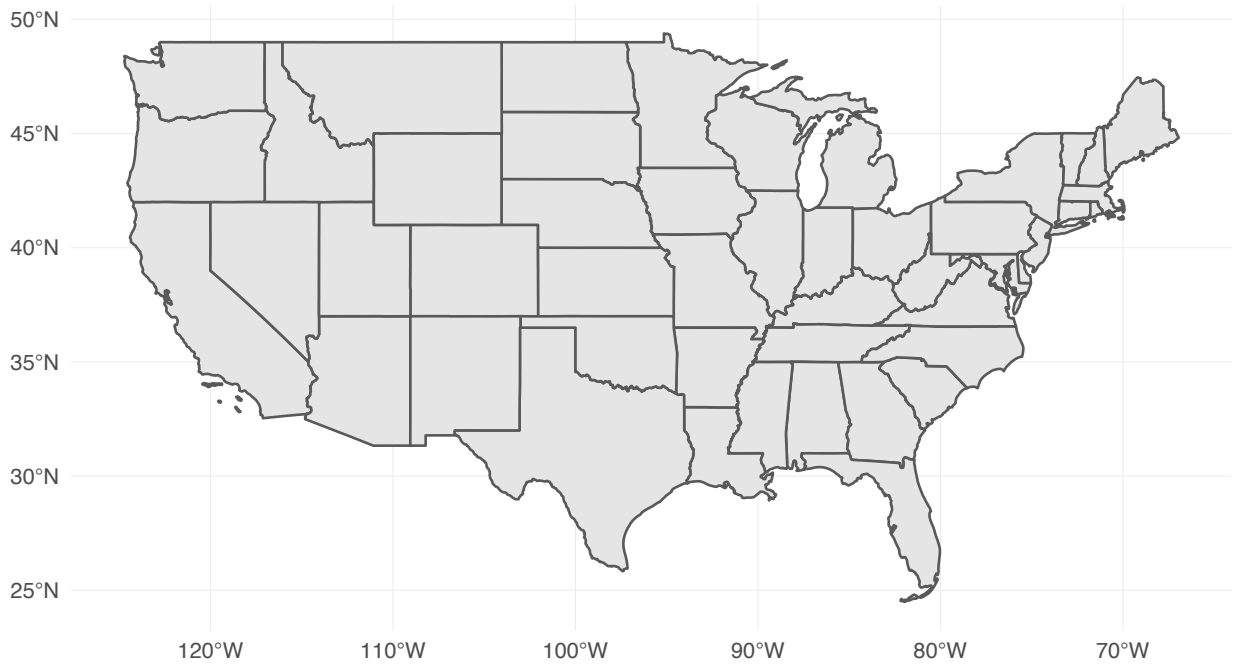
Now, we have two new variables `lng` and `lat` in the data set.

Next, we create a base map object for 48 states. For this we need to obtain the shape files from the US Census Bureau. Download them from here: https://www.census.gov/geo/maps-data/data/cbf/cbf_state.html and save on your computer.

```
usa_48 <- sf::st_read(here::here("cb_2017_us_state_20m.shp")) %>%
  filter(!(NAME %in% c("Alaska",
                       "District of Columbia",
                       "Hawaii",
                       "Puerto Rico")))
```

Plot these using `ggplot`

```
ggplot(data = usa_48) +
  geom_sf() +
  theme_minimal()
```

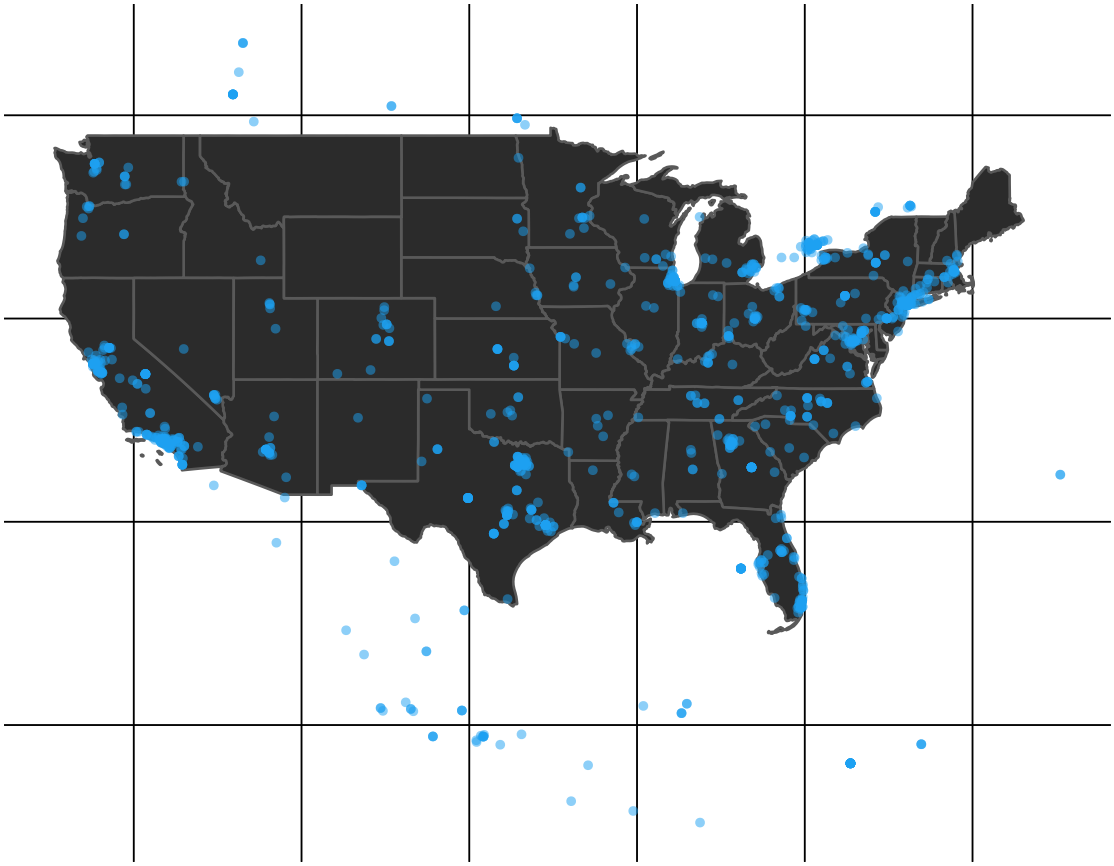


Now, we will overlay the Twitter data on top of the US map. For this, we will have to convert it into an `sf` object.

```
lp_geo_sf <- st_as_sf(filter(lp_geo, !is.na(lat)),
                      coords = c("lng", "lat"))
st_crs(lp_geo_sf) <- 4326   # set the coordinate reference system
```

Now we are ready to make the plot!

```
ggplot() +
  geom_sf(data = usa_48, fill = "#2b2b2b") +
  geom_sf(data = lp_geo_sf,
          shape = 16,
          alpha = 0.5,
          color = "#1da1f2") +
  theme_minimal() +
  theme(axis.text.x = element_blank(),
        axis.text.y = element_blank(),
        panel.grid.major = element_blank())
```



You may try to find some pattern here, but in my experience, Twitter activity is pretty much correlated with the population.

5.6 Sentiment analysis

We will do some basic sentiment analysis. The objective is to find out the general sentiment in our tweets. The variable of interest here is `text`, which has all the tweet text. We will use lexicon-based method to identify the sentiment in each tweet first and then we will aggregate them all.

Table 5.1: Twitter Sentiment

anger	anticipation	disgust	fear	joy	sadness	surprise	trust	negative	positive
0	2	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0
0	1	0	0	1	0	1	1	0	2
0	0	0	1	0	1	0	3	1	2
0	0	0	0	0	0	0	0	0	0

For this, we will use `get_nrc_sentiment()` function from `syuzhet` package. Note that the execution takes some time so please be patient.

```
lp_sent <- lp$text %>%
    syuzhet::get_nrc_sentiment()
```

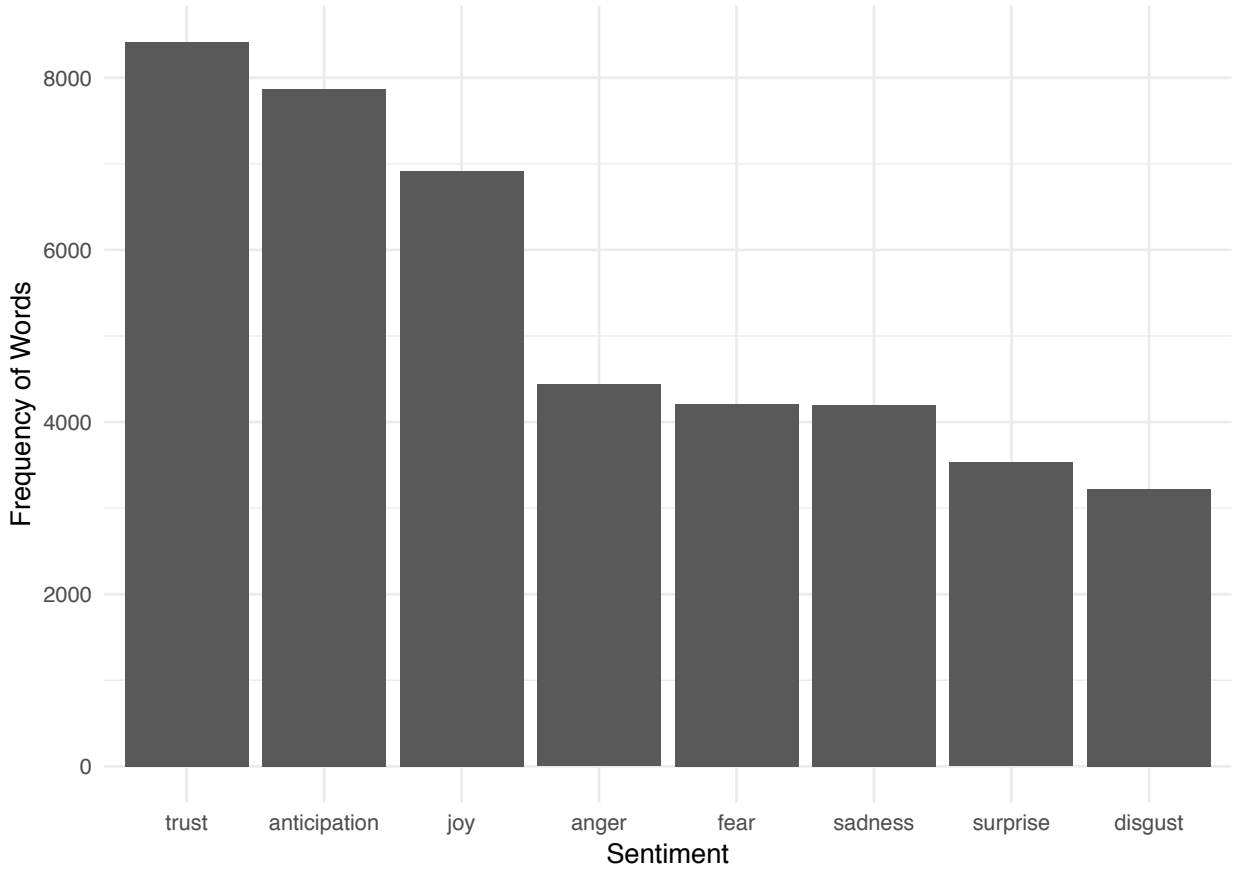
Take a peek at the data

```
head(lp_sent) %>%
  knitr::kable(caption = "Twitter Sentiment")
```

In the above table, the numbers mean the number of words in each tweet that fall into that specific sentiment category. So the first tweet has 2 words indicating anticipation and fifth tweet has 3 words indicating trust. It's better to aggregate the sentiments and plot them for easy interpretations.

```
lp_sent %>%
  summarize_all(sum, na.rm = TRUE) %>%
  select(-negative, -positive) %>% # Dropping these helps in plotting
  reshape2::melt() %>%
  ggplot(aes(reorder(variable, -value), value)) +
  geom_col() +
  labs(x = "Sentiment", y = "Frequency of Words") +
  theme_minimal()
```

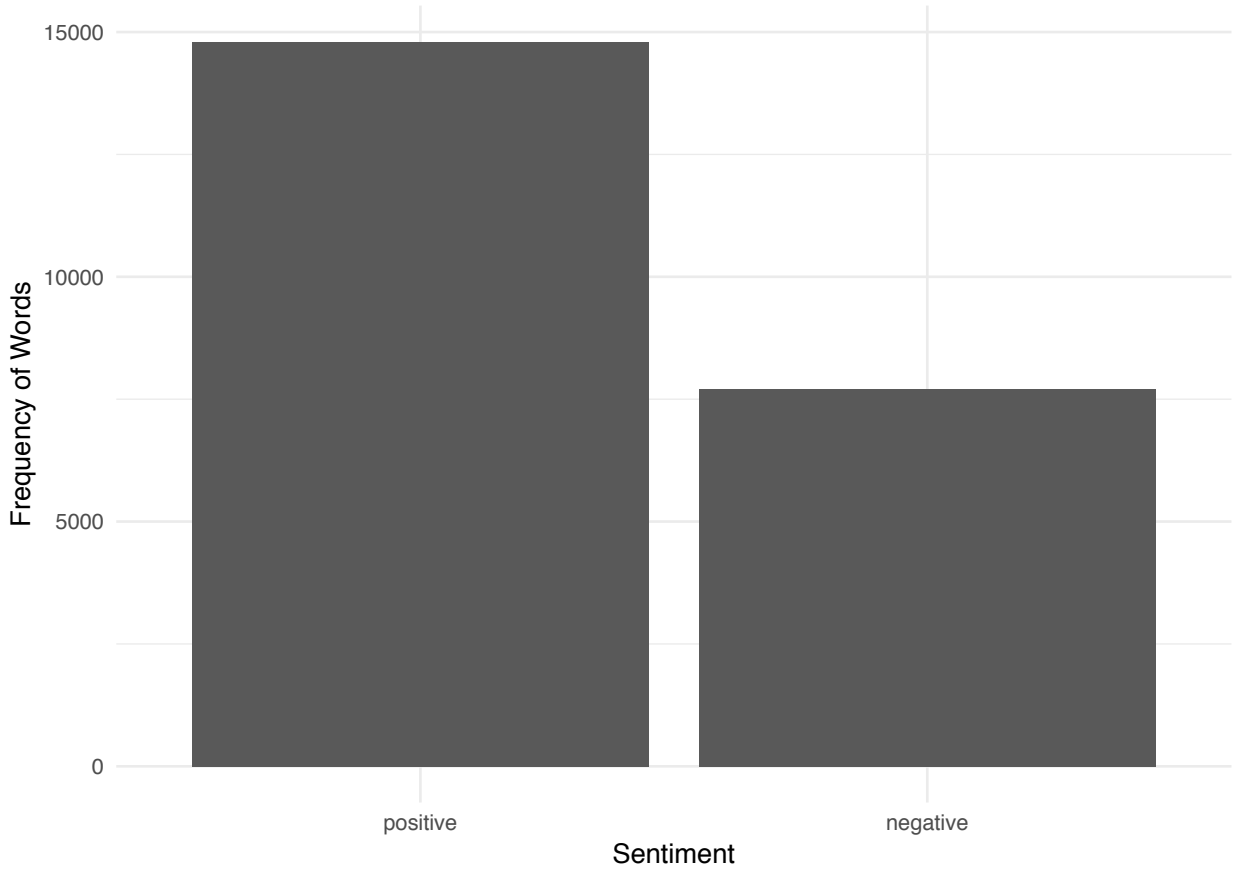
```
## No id variables; using all as measure variables
```



Plot only positive and negative sentiment.

```
lp_sent %>%
  summarize_all(sum, na.rm = TRUE) %>%
  select(negative, positive) %>%
  reshape2::melt() %>%
  ggplot(aes(reorder(variable, -value), value)) +
  geom_col() +
  labs(x = "Sentiment", y = "Frequency of Words") +
  theme_minimal()
```

```
## No id variables; using all as measure variables
```



Certainly, the tweets have a lot of positive sentiment! But does that matter to make tweets more popular?

5.6.1 Linear regression

In order to find out whether the sentiment can affect the count of favorites, we will do linear regression analysis. We will regress log of `favorite_count` on the sentiment counts as well as whether the tweets is verified and log of `followers_count`. As both the counts can be 0, we add 1 to them before taaking the log.

```
cbind(lp, lp_sent) %>%
  mutate(favorite_count = favorite_count + 1) %>%
  lm(log(favorite_count) ~ anger + anticipation + disgust + fear + joy +
               sadness + surprise + trust + verified + log(followers_count + 1),
             data = .) %>%
  summary()
```

```
##
## Call:
## lm(formula = log(favorite_count) ~ anger + anticipation + disgust +
##     fear + joy + sadness + surprise + trust + verified + log(followers_count +
```

```
##     1), data = .)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.9398 -0.5172 -0.2173  0.3306  6.6115
##
## Coefficients:
##                             Estimate Std. Error t value Pr(>|t|)
## (Intercept)               -0.496412   0.023189 -21.407  < 2e-16 ***
## anger                      0.011007   0.016752   0.657 0.511160
## anticipation              -0.008309   0.012225  -0.680 0.496708
## disgust                    0.010000   0.018799   0.532 0.594749
## fear                       0.020153   0.016476   1.223 0.221276
## joy                        0.065785   0.013607   4.835 1.34e-06 ***
## sadness                   -0.007841   0.016786  -0.467 0.640449
## surprise                  -0.048582   0.015825  -3.070 0.002144 **
## trust                      0.033538   0.009916   3.382 0.000721 ***
## verifiedTRUE               1.029077   0.039416  26.108  < 2e-16 ***
## log(followers_count + 1)   0.169462   0.003829  44.257  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.7896 on 17894 degrees of freedom
## Multiple R-squared:  0.1976, Adjusted R-squared:  0.1971
## F-statistic: 440.5 on 10 and 17894 DF,  p-value: < 2.2e-16
```

So, it looks like joyful and trusting tweets are favorited a lot while tweets with surprise are less favorited. Note that I have controlled for the follower count as well as whether the account was verified. Both these variables are highly significant.[4]

5.7 Create a wordcloud

Wordcloud is a popular visualization tool, yet I am not a big fan of it. However, it seems that managers love a wordcloud because they can get the main message in one quick look. We will create a wordcloud using `eom_text_wordcloud()` function from `ggwordcloud` package.

Before we can make a wordcloud, there is some preprocessing of the text that is necessary. First of all, we need to tokenize the text so that we have all the words separately identified. Next, we get rid of all the "stop words" such as articles (e.g., the, an), pronouns (e.g., he, she, it), etc. We also need to remove other words that we think may contaminate the wordcloud.[5]

Create a tibble of all the words we want to get rid of. This list needs to be updated depending on what shows up in the wordcloud below.

[4] As an exercise, rerun the above regression using `retweet_count` as the dependent variable.
[5] This requires some trial and error.

```
exclude_words <- tibble(word = c("http", "https", "twitter", "t.co",
                                 "liverpool", "barcelona", "barca"))
```

We have to first get the words from all the tweets

```
word_tokens <- lp_geo %>%
  select(status_id, text) %>%
  tidytext::unnest_tokens(word, text) %>%
  anti_join(stop_words) %>%
  anti_join(exclude_words)
```

```
## Joining, by = "word"
## Joining, by = "word"
```

```
head(word_tokens)
```

```
## # A tibble: 6 x 2
##   status_id           word
##   <chr>               <chr>
## 1 1125931671986028544 poznaninmypants
## 2 1125931671986028544 destnd4gr8tnes2
## 3 1125931671986028544 yankeegunner
## 4 1125931671986028544 gunnerblog
## 5 1125931671986028544 clivepafc
## 6 1125931671986028544 vision
```

The first few words that we see are probably hashtags this user used. We don't need to pay attention to individual words at this point.

Find the frequency of each word and then rank them in descending order

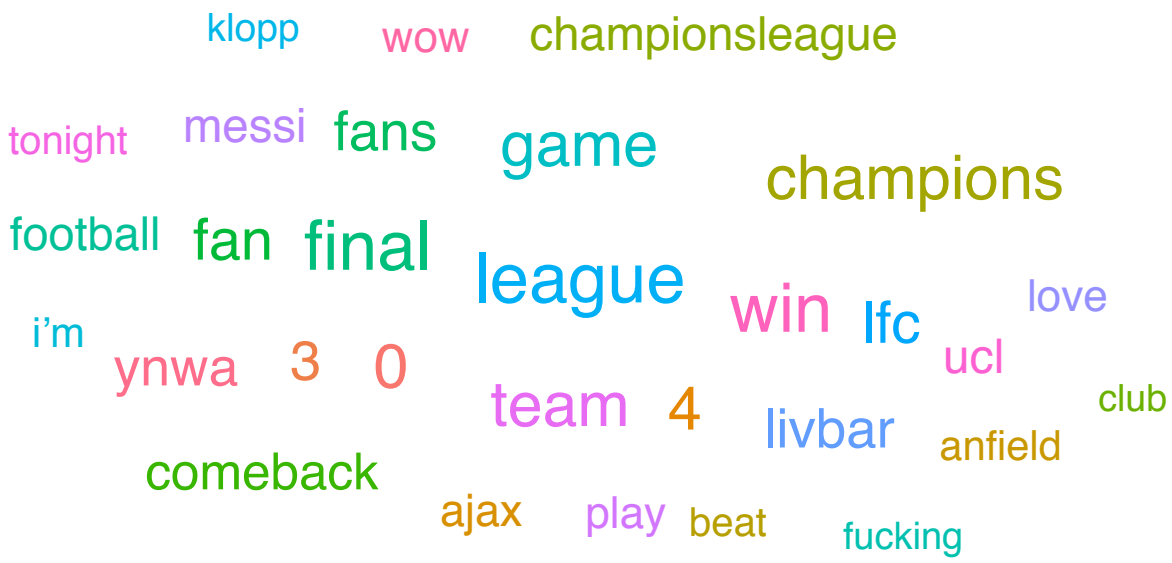
```
word_tokens_count <- word_tokens %>%
  count(word, sort = TRUE)

head(word_tokens_count)
```

```
## # A tibble: 6 x 2
##   word       n
##   <chr>  <int>
## 1 league  1692
## 2 final   1647
## 3 win     1599
## 4 game    1358
## 5 team    1335
## 6 0       1330
```

Make the wordcloud

```
set.seed(2019)

word_tokens_count %>%
  top_n(30) %>%
  ggplot(aes(label = word, size = n, color = word)) +
  scale_size_area(max_size = 10) +
  geom_text_wordcloud() +
  theme_minimal()
```

```
## Selecting by n
```



Another way you can create wordcloud quickly is using a custom R function available from STHDA

```
source('http://www.sthda.com/upload/rquery_wordcloud.r')
```

Full wordcloud. The default is to plot 200 words.

```r
rquery.wordcloud(x = lp$text, type = "text")
```

Wordcloud without the excluded words:

```r
rquery.wordcloud(x = lp$text, type = "text",
                 excludeWords = c("http", "https", "twitter", "t.co",
                                  "liverpool", "barcelona", "barca"),
                 max.words = 50)
```

5.8 Summary

In this exercise we saw how to download data from Twitter, plot it on the map, perform basic sentiment analysis, and make a wordcloud. There are numerous other exercises that are possible using social media data. Other popular APIs that you can use are for Yelp, Reddit, FourSqaure, etc. Check out their documentation and see whether you can get data from these services.

Chapter 6

Airlines Customer Satisfaction

Just over one year ago I wrote a blog post titled "Customer Satisfaction of American Airline Companies" on Wordpress.com.[1] In that post I compared Twitter sentiment of a few American airlines to the customer satisfaction scores reported by University of Michigan's American Customer Satisfaction Survey (ACSI). I found that the correlation between Twitter sentiment and ACSI was 0.77. I did not report rank correlation but I recall that it was about the same or slightly lower.

In this chapter we will recreate this analysis.

6.1 American Customer Satisfaction Index

Before we proceed with the analysis, it is important to understand what ACSI is and how it is calculated. I strongly recommend reading this brief description on their website. Marketing academicians, including yours truly[2], have extensively used ACSI in their research.

Even though it is very popular, ACSI has many drawbacks. One major drawback is that for any given brand it is reported only once a year. However, customer satisfaction may change a lot even from month to month. As Twitter data is easily available, perhaps companies can use it instead of ACSI.

6.2 Tasks to complete

In this exercise we will complete following tasks:

1. Download tweets on 9 American airlines which are covered by ACSI.
2. Perform sentiment analysis on the tweets.
3. Correlate Twitter sentiment scores with ACSI and report the results.

[1] https://ashwinmalshe.wordpress.com/2016/04/03/customer-satisfaction-of-american-airline-companies/
[2] https://www.ashwinmalshe.com/files/Malshe_Agarwal_2015.pdf

Table 6.1: Airlines Customer Satisfaction

Airline	Twitter Handle	ACSI Score
Alaska	\@AlaskaAir	80
Southwest	\@SouthwestAir	79
JetBlue	\@JetBlue	79
Delta	\@Delta	75
American	\@AmericanAir	73
Allegiant	\@Allegiant	71
United	\@United	70
Frontier	\@FlyFrontier	64
Spirit	\@SpiritAirlines	63

6.3 Download tweets

Start with loading Twitter credentials in your R session and loading relevant packages as shown below. For instructions on getting a Twitter token, please see Chapter 5.2.

```
library(rtweet)  # Twitter package
library(dplyr)
library(ggplot2)
library(reshape2)
library(purrr)
library(janitor) # Row percentages
library(Hmisc) # Correlations
library(ggcorrplot) # Correlations plot

# Packages for text analysis
library(syuzhet)
```

Load Twitter token

```
load(here::here("twitter_token"))
```

Take a look at the ACSI scores of airlines.[3]

Table 6.1 shows Twitter handles for the airlines. I also copied the 2019 scores and pasted in this table.

From Table 6.1, Alaska Airlines has the highest customer satisfaction while Frontier and Spirit have the lowest customer satisfaction. Both these airlines are low cost and people constantly complain about them.[4]

[3] https://www.theacsi.org/acsi-benchmarks/benchmarks-by-industry
[4] Check out the reviews of Frontier on TripAdvisor.

6.3.1 Collect tweets

In the following code, we first create a vector `airline_tw` which has Twitter handles for the 9 airlines. Next we set up an empty list `airlines_list` to hold the tweets for each airline. The critical piece of code is the `for` loop. We will download up to 2,000 tweets per airline. You can try to download more if you want. I just wanted to stay within the rate limit and get all the tweets at once.[5] We also limit the language of the tweets to English and geography to the US.

The output of the following code will be `airlines_list` with 9 data frames with a maximum of 2,000 rows in any data frame.[6]

```
airline_tw <- c("@AlaskaAir", "@SouthwestAir", "@JetBlue",
                "@Delta", "@AmericanAir", "@Allegiant",
                "@United", "@FlyFrontier", "@SpiritAirlines")

airlines_list <- list()

for (i in 1:9) {
  print(paste("Getting tweets for", airline_tw[i]))

  airlines_list[[i]] <- search_tweets(
    q = airline_tw[i],
    lang = 'en',
    geocode = lookup_coords("usa"),
    n = 2000,
    include_rts = FALSE, # exclude retweets
    )
}
```

6.3.2 Adding airline as a column

Ideally we would like to stack 9 data frames on top of each other and then carry out the sentiment analysis. However, none of the data frames has a column that identifies which airline the tweets belong to! I strongly encourage you to take a look at any of the 9 data frames by using `names()` and `head()` functions.

In order to add a column in each data frame while still being a part of the list, we will use `map2_dfr()` function from `purrr` package. This function iterates over two arguments simultaneously and then row binds the resulting data frames. In the code below, it will iterate over the list `airlines_list` while also iterating over the vector `Airline`. Note that `Airline` just holds the names of the 9 airlines. `map2_dfr()` will then add (using `mutate()`) a column called `airline` to each data frame stored in `airlines_list` and assign this column the value stored in the vector `Airline`.[7] Finally, it will row bind these 9 data frames and return a single data frame called `airlines_df`.

[5] Recall that Twitter allows you to download 18,000 tweets every 15 minutes.

[6] It will take about 5-10 minutes depending on your Internet speed.

[7] If you find this confusing, you need to read more on `map()` family of functions from `purrr`.

Table 6.2: Airlines Sentiment

anger	anticipation	disgust	fear	joy	sadness	surprise	trust	negative	positive
0	2	0	0	0	0	0	0	0	0
1	1	0	0	1	0	1	1	1	1
0	1	0	0	1	0	0	1	0	1
0	0	0	0	0	0	0	0	0	0
0	2	0	2	0	1	0	1	2	1
1	2	1	0	1	1	1	2	1	2
0	1	0	0	1	0	0	1	0	1
0	1	0	0	0	0	0	0	0	0

```
Airline = c("Alaska", "Southwest", "JetBlue",
            "Delta", "American", "Allegiant",
            "United", "Frontier", "Spirit")

airlines_df <- map2_dfr(.x = airlines_list,
                        .y = Airline,
                        ~ mutate(.x, airline = .y) )
```

6.4 Sentiment analysis

Now that we have assembled the data set with tweets pertaining to the 9 airlines, we are ready to do sentiment analysis. We will use `get_nrc_sentiment()` from `syuzhet` package. The input to this function is a character vector. Therefore, we will simply `pull()` this vector out from `airlines_df`.

Depending on the number of tweets this code will take a few minutes to execute so please be patient.

```
airlines_sent <- airlines_df %>%
  pull(text) %>%  # This returns a character vector
  get_nrc_sentiment()
```

Take a look at the sentiment data using `head()`. For my sample, the results are shown in Table 6.2,

6.5 Net Sentiment Score (NSS)

In this step, we will aggregate the sentiment at airline level so that we will have just 1 observation for every airline. However, note that `airlines_sent` does not have any column identifying the airline. This is because we used only the `text` column from that data set. In the code below, we will first add back some of the relevant variables using `cbind()`. The variables of interest are `airline`, `favorite_count`, and `retweet_count`. We will retain `favorite_count`, and `retweet_count` because they can be used as weights.

In the code below, I have commented the blocks. They are self explanatory. The last block where we calculate the net sentiment scores (NSS) needs some explanation. NSS are similar to the (in)famous Net Promoter Score (NPS).[8] The idea is that we take the difference between the positive sentiment and negative sentiment scores and divide this difference by the total tweets (or sum of weights for weighted metric). The NSS formula in general for our case is as follows:

$$NSS = \frac{\sum w_i.PS_i - \sum w_i.NS_i}{\sum w_i}$$

where, w_i is the weight assigned to the tweet (i.e., number of favorites or retweets), PS_i is the positive sentiment score of a given tweet, and NS_i is the negative sentiment score of a given tweet. For raw NSS, where we do not weight by number of favorites or retweets, $w_i = 1 \quad \forall i$

```
There is no reason to believe that NSS will be correlated strongly with ACSI.
However, in my blog post it did and here we are assessing whether that
relationship still holds.
```

```
airlines_final <- cbind(
  airlines_df %>% select(airline, favorite_count, retweet_count),
  airlines_sent %>% select(negative, positive)
  ) %>%
  # Create new "weighted" variables
  mutate(negative_fav = negative * favorite_count,
         positive_fav = positive * favorite_count,
         negative_rt  = negative * retweet_count,
         positive_rt  = positive * retweet_count) %>%
  # Get the sum of these variables for each airline
  group_by(airline) %>%
    summarise(neg_sum     = sum(negative),
              neg_fav_sum = sum(negative_fav),
              neg_rt_sum  = sum(negative_rt),
              pos_sum     = sum(positive),
              pos_fav_sum = sum(positive_fav),
              pos_rt_sum  = sum(positive_rt),
              fav_sum     = sum(favorite_count),
              rt_sum      = sum(retweet_count),
              tot_obs     = n()) %>%
  ungroup() %>%
  # Calculate sentiment metrics
  mutate(nss     = (pos_sum - neg_sum) / tot_obs,
         nss_fav = (pos_fav_sum - neg_fav_sum) / fav_sum,
         nss_rt  = (pos_rt_sum - neg_rt_sum) / rt_sum) %>%
  # Add the column of customer satisfaction
  mutate(acsi = c(80, 71, 73, 75, 64, 79, 79, 63, 70))
```

[8] Read more here

Table 6.3: Sentiment and ACSI Correlations

	nss	nss_fav	nss_rt	acsi
nss	1.000	0.528	0.518	0.138
nss_fav	0.528	1.000	0.857	-0.297
nss_rt	0.518	0.857	1.000	-0.251
acsi	0.138	-0.297	-0.251	1.000

6.6 The moment of truth

Now comes the final stage where we check the correlations between various NSS measures and ACSI. For this I use `ggcorplot()` function from `ggcorplot` package. As this is not a major topic for this exercise, I leave the explanation of the code to you as an exercise.

```
ggcorrplot::ggcorrplot(
  airlines_final %>%
    select(starts_with("nss"), acsi) %>%
    cor() %>%
    round(2),
  p.mat = ggcorrplot::cor_pmat(
    airlines_final %>%
      select(starts_with("nss"), acsi)
    ),
  hc.order = TRUE,
  type = "lower",
  outline.color = "white",
  ggtheme = ggplot2::theme_minimal,
  colors = c("#cf222c", "white", "#3a2d7f")
  )
```

From Figure 6.1 it looks like ACSI has somewhat negative correlations with each of the NSS metric! This is not good news...for ACSI! :)Furthermore, the crosses on the squares indicate statistical non-significance. However, as I explain below, we will do a better comparison with more direct sentiment metrics.

Table 6.3 shows the correlations in numbers. Indeed, ACSI is marginally negatively correlated with NSS metrics.

6.7 Correlating with granular sentiments

Thus far we used only positive and negative sentiments. However, we actually have much granular sentiment scores in the data. Let's check whether these scores do a better job of explaining the pattern in the data.

For this, we will simply use the percentage of words with a specific sentiment in a tweet. For instance, if there were 2 words that were labeled as "joy" by `syuzhet` out of the 5 words it labeled overall from a tweet, we consider it is 40% (2/5) joy. It's not the cleanest metric but it will work.

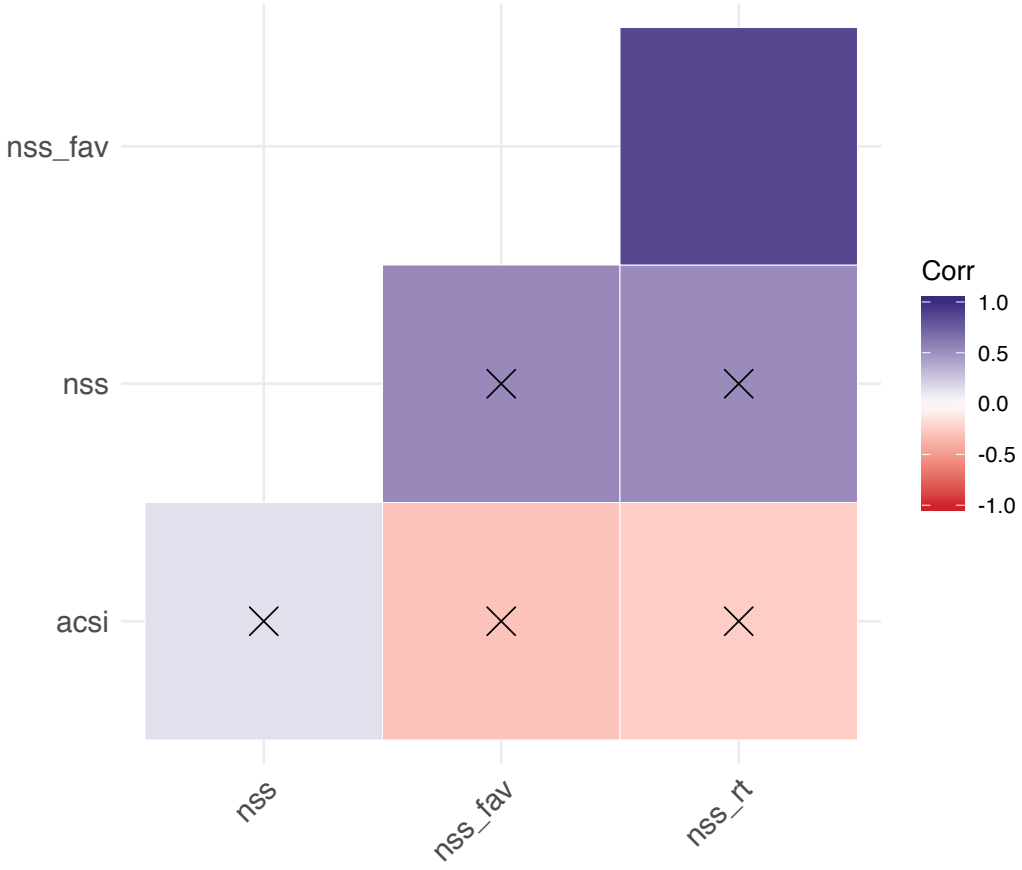


Figure 6.1: Correlation Plot

To calculate row percentages, we will use `adorn_percentages()` function from `janitor` package. This function has two drawbacks. First, it assumes that the first column is "id" column and it doesn't take it into account for row calculations. We overcome this problem by adding a column of airlines and then making it the first column using `select()` function from `dplyr`. Second, the package returns `NaN` when the row sums are 0. This is not a drawback in general but our application needs a 0 in place of `NaN`. We will fix this using `is.na()` function from base R.

```
sent_cor <- airlines_sent %>%
    # Add airline names
  mutate(airline = airlines_df$airline) %>%
  select(airline, everything(), -c(positive, negative)) %>%
  janitor::adorn_percentages() %>%
  as.data.frame()

# Replace NaN with 0

sent_cor[is.na(sent_cor)] <- 0

# Finally summarize and add ACSI

sent_cor <- sent_cor %>%
  group_by(airline) %>%
  summarize_if(is.numeric, mean) %>%
  # Add ACSI scores
  mutate(acsi = c(80, 71, 73, 75, 64, 79, 79, 63, 70)) %>%
  select(-airline)
```

6.7.1 Correlation plot

Figure 6.2 shows the correlation plot.

```
ggcorrplot::ggcorrplot(
  sent_cor %>%
  cor(method = "pearson") %>%
  round(3),
  p.mat = ggcorrplot::cor_pmat(sent_cor, method = "pearson"),
  hc.order = TRUE,
  type = "lower",
  outline.color = "white",
  ggtheme = ggplot2::theme_minimal,
  colors = c("#cf222c", "white", "#3a2d7f")
  )
```

ACSI has positive correlations with joy, surprise, and anticipation. It has negative correlations

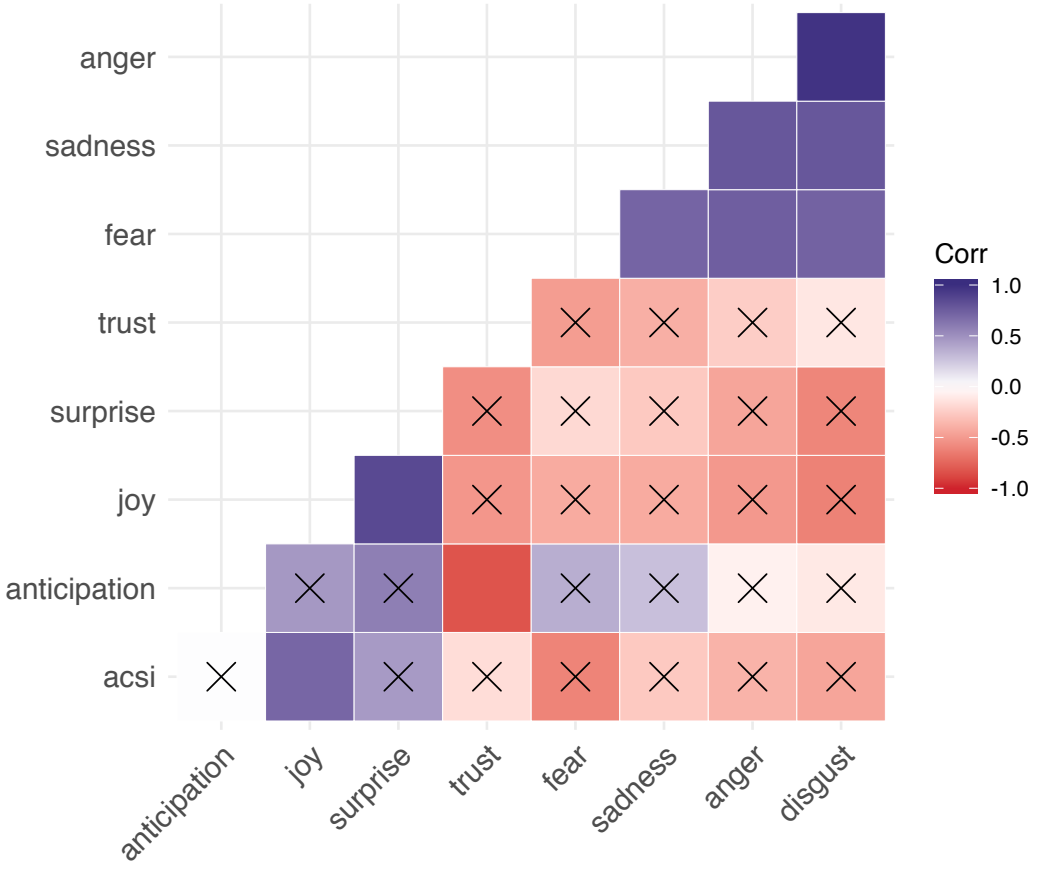


Figure 6.2: Granual Sentiment Correlation Plot

Table 6.4: Granular Sentiment and ACSI Correlations

	anger	anticipation	disgust	fear	joy	sadness	surprise	trust	acsi
anger	1.000	-0.073	0.966	0.738	-0.513	0.783	-0.449	-0.250	-0.377
anticipation	-0.073	1.000	-0.111	0.357	0.462	0.278	0.585	-0.833	0.007
disgust	0.966	-0.111	1.000	0.721	-0.622	0.777	-0.598	-0.122	-0.450
fear	0.738	0.357	0.721	1.000	-0.418	0.714	-0.193	-0.490	-0.611
joy	-0.513	0.462	-0.622	-0.418	1.000	-0.423	0.848	-0.523	0.700
sadness	0.783	0.278	0.777	0.714	-0.423	1.000	-0.269	-0.398	-0.274
surprise	-0.449	0.585	-0.598	-0.193	0.848	-0.269	1.000	-0.561	0.449
trust	-0.250	-0.833	-0.122	-0.490	-0.523	-0.398	-0.561	1.000	-0.166
acsi	-0.377	0.007	-0.450	-0.611	0.700	-0.274	0.449	-0.166	1.000

with the rest. Surprisingly, it has a negative correlation with trust.[9] Unfortunately, none of these correlations is statistically significant at 5% level of significance! This is somewhat expected because we have only 9 airlines.

6.7.2 Correlation matrix

Let's take a look at the correlations as shown in Table 6.4. We will also output the p values this time. For this, we will use `rcorr()` function from `Hmisc` package. `rcorr()` outputs a list with correlations and their p values in separate matrices.

```
sent_cor %>%
  as.matrix() %>%
  Hmisc::rcorr() %>%
  .$r %>%
  round(3)
```

Table 6.5 shows the p values corresponding to the correlation coefficients. The p value for the correlation coefficient between ACSI and joy is significant at 10% level.

```
sent_cor %>%
  as.matrix() %>%
  Hmisc::rcorr(type = "pearson") %>%
  .$P %>%
  round(3)
```

6.8 Summary

In this chapter, we analyzed the correlations between Twitter sentiment and customer satisfaction of 9 American airlines. We used American Customer Satisfaction Index (ACSI) as the measure of customer satisfaction. We find that there is little correlation between the two metrics.

[9] Any speculations for this result?

Table 6.5: p Values for Granular Sentiment and ACSI Correlations

	anger	anticipation	disgust	fear	joy	sadness	surprise	trust	acsi
anger	NA	0.852	0.000	0.023	0.158	0.013	0.226	0.517	0.318
anticipation	0.852	NA	0.777	0.346	0.211	0.470	0.098	0.005	0.986
disgust	0.000	0.777	NA	0.028	0.074	0.014	0.089	0.754	0.224
fear	0.023	0.346	0.028	NA	0.263	0.031	0.619	0.180	0.080
joy	0.158	0.211	0.074	0.263	NA	0.257	0.004	0.148	0.036
sadness	0.013	0.470	0.014	0.031	0.257	NA	0.484	0.289	0.476
surprise	0.226	0.098	0.089	0.619	0.004	0.484	NA	0.116	0.225
trust	0.517	0.005	0.754	0.180	0.148	0.289	0.116	NA	0.670
acsi	0.318	0.986	0.224	0.080	0.036	0.476	0.225	0.670	NA

However, ACSI is measured only once annually while Twitter sentiment can be obtained every single day. Furthermore, ACSI is a measure of customer satisfaction. Twitter sentiments that we used do not necessarily say anything about satisfied customers. It could be a good metric for brand attitude instead.

Chapter 7

Event Study

Event studies are popular in finance research. The method is old and yet quite robust even today. The core assumption underlying an event study is that the stock markets are "informationally efficient". Current stock prices reflect all the publicly available information and the stock prices adjust quickly to the release of all new public information that is value relevant.[1] If you want to know more about market efficiency, I suggest reading this article: https://www.investopedia.com/terms/s/semistrongform.asp

7.1 Mechanics of event studies

An event study looks at an event that was unexpected and quantifies the impact of that event on the stock prices. The event will be value relevant only if it moves the stock price more than what was expected by the market. In order to determine whether an event is value relevant, we need to have a benchmark against which we can measure the stock price movement.

7.1.1 The benchmark

The most common benchmark used in finance is the Fama-French (FF) 3-factor asset pricing model. The model is a modified version of more famous and theory-rich Capital Asset Pricing Model (CAPM).[2] FF model asserts that the stock prices are determined by 3 risks: market risk, size risk, and value risk. FF model for a stock i in time t is given by

$$ExcessRet_{it} = \beta_{MKT} * MktRF_{it} + \beta_{SMB} * SMB_{it} + \beta_{HML} * HML_{it}$$

where, $ExcessRet_{it}$ is the stock's return above the risk-free return, $MktRF_{it}$ is the market return above risk-free return, SMB_{it} is the "size factor", and HML_{it} is the "value factor".[3]

[1] This is known as semi-strong form of market efficiency.

[2] William Sharpe won the Nobel Memorial Prize in Economics for CAPM.

[3] We will not get into the details of these factors but you can read more about them from Kenneth French's website.

β_{MKT}, β_{SMB}, and β_{HML} are the measures of market, size, and value risks, respectively. As long as we have estimates of these risks[4], we can use them to obtain **expected** stock returns in time $t+k$ by observing $MktRF_{it+k}$, SMB_{it+k}, and HML_{it+k} as follows:

$$\widehat{ExcessRet}_{it+k} = \hat{\beta}_{MKT} * MktRF_{it+k} + \hat{\beta}_{SMB} * SMB_{it+k} + \hat{\beta}_{HML} * HML_{it+k}$$

Thus, $\widehat{ExcessRet}_{it+k}$ is our benchmark against which we will compare the realized stock returns. If the realized returns are statistically not distinct from the expected returns, we conclude that the event was not value relevant.

7.2 Steps for an event study

We have to perform the following steps:

1. Obtain estimates of the risks by regressing daily stock returns on the risk factors. Usually we use returns from the past 250 trading days (1 calendar year).
2. Use the risk estimates and risk factors on the day of the event to estimate expected stock returns.
3. Test whether the realized stock returns are statistically different from expected stock returns. Note that the expected stock returns will be estimated with their prediction intervals. As long as the realized stock returns do not fall in the prediction interval, we can conclude that the event is likely to be value relevant.

We now turn to doing an event study analysis.

7.3 Case study of Donald Trump's Twitter attacks

The current US President, Donald Trump, often attacks companies on Twitter. Many people object to this because they fear that the companies' stock prices will be affected adversely. However, whether Trump's Twitter attack are seriously damaging to the stock prices is an empirical question. Assuming that Trump's attacks are unexpected events, we can use event study methodology to answer it.

Figure 7.1 is a sample of Trump's attack on Amazon.

I used a CNN article that listed 17 such attacks as of April 2018. I picked 13 of these for the event study. I left out a few cases where the company under attack was not traded publicly but their parent company was.

Based on this, let's create a data frame that will hold the 13 events.

[4] Call them $\hat{\beta}_{MKT}$, $\hat{\beta}_{SMB}$, and $\hat{\beta}_{HML}$

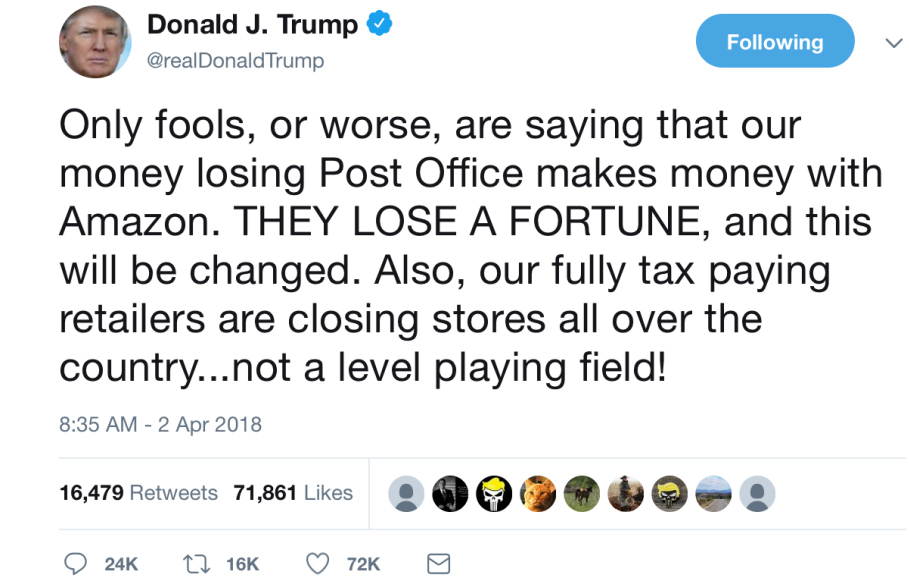


Figure 7.1: Trump Tweet Attacking Amazon

```
events <- data.frame(Company = c("Amazon", "Boeing", "CBS", "Comcast",
                        "Delta", "Facebook", "General Motors",
                        "Lockheed Martin", "Merck", "New York Times",
                        "New York Times", "Nordstrom", "Toyota"),
           Ticker = c("AMZN", "BA", "CBS", "CCZ", "DAL", "FB",
                      "GM", "LMT", "MKGAF", "NYT", "NYT", "JWN", "TM"),
           Date_Attacked = as.Date(c("2018-04-02", "2016-12-06",
                                     "2017-02-17", "2017-11-29",
                                     "2017-01-30", "2017-10-21",
                                     "2017-01-03", "2016-12-22",
                                     "2017-08-14", "2017-02-17",
                                     "2017-01-28", "2017-02-08",
                                     "2017-01-05")))
```

These 13 events are listed in Table 7.1.

7.4 Doing the event study

Start with loading the required packages. In order to get stock data, we will use a specialized package called `BatchGetSymbols` available on CRAN.

```
library(BatchGetSymbols)
library(dplyr)
```

Table 7.1: Trump Twitter Attacks

Company	Ticker	Date_Attacked
Amazon	AMZN	2018-04-02
Boeing	BA	2016-12-06
CBS	CBS	2017-02-17
Comcast	CCZ	2017-11-29
Delta	DAL	2017-01-30
Facebook	FB	2017-10-21
General Motors	GM	2017-01-03
Lockheed Martin	LMT	2016-12-22
Merck	MKGAF	2017-08-14
New York Times	NYT	2017-02-17
New York Times	NYT	2017-01-28
Nordstrom	JWN	2017-02-08
Toyota	TM	2017-01-05

```
library(ggplot2)
library(lubridate)
library(data.table)
library(purrr)
library(psych)
library(reshape2)
library(caret)
```

Download stock price data using the `events` data frame. We specifically need the ticker symbols of the stocks.[5] We will collect stock returns 400 days prior to the event and 15 days post event. We are not going to use all the data but if you are downloading it once, it is better to download more rather than less.

The function `BatchGetSymbols()` returns a list. One of the elements of the list is the data frame consisting of the stock prices. Rather than writing complicated code in the loop, we will simply store the entire list as an element of a larger list `stkprc`. Then we will keep storing downloaded lists in this larger list.

```
stkprc <- list()

for (i in 1:nrow(events)) {
  stkprc[[i]] <- BatchGetSymbols(
                  tickers = events$Ticker[i],
                  first.date = (events$Date_Attacked[i] - 400),
                  last.date = (events$Date_Attacked[i] + 15),
                  freq.data = "daily"
                  )
}
```

[5] A ticker symbol is a short code used for identifying a company's stock.

`stkprc` is a list of lists as each one of its elements is a list of 2 more elements. Next we need to create another list `price_dt` where we will only hold the dataframe with stock prices. Also, we need to create a variable that holds the information about the trading days from the event date. This is slightly more complicated than just taking the difference between two dates. This is because the stock market is not open on all the days. In the code below, I used `which()` function from base R to return the row number where the event date rests in the downloaded stock returns data. This has to be done for each of the 13 stocks and therefore, we use a `for` loop. Note that we create a variable `event_diff` which is the difference between the row number of a given date and the row number of the event date that we calculated earlier.

```
price_dt <- list()

for (i in 1:nrow(events)) {
  event_pos <- which(stkprc[[i]]$df.tickers$ref.date ==
                       events$Date_Attacked[i])
  price_dt[[i]] <- tryCatch(stkprc[[i]]$df.tickers %>%
    mutate(event_diff = row_number() - event_pos),
    error = function(x) {
    message(x)
    return(NA)
    })
}
```

You probably noticed that there is a `tryCatch()` function used above. As it turns out, there were 2 events which did not take place on the day when stock markets were open. This means for these stocks, `event_pos` was `NA` and `event_diff` could not be computed. We could move the event date for these two stocks by a day or two. However, for this exercise we will refrain from doing that because there might be other events happening the next day, which will affect the stock prices.

The next two lines of code will remove the blank list elements.

```
price_dt[[which(is.na(price_dt))[1]]] <- NULL
price_dt[[which(is.na(price_dt))[1]]] <- NULL
```

Finally, we will stack all the 11 data sets and create one single data frame.[6]

```
price_stakced <- data.table::rbindlist(price_dt)
```

Get the frequency distribution of tickers to make sure that we actually stacked 11 data sets.

```
table(price_stakced$ticker)
```

```
##
##  AMZN    BA   CBS   CCZ   DAL    GM   JWN   LMT MKGAF   NYT    TM
##   286   287   286   287   285   285   286   285   287   286   285
```

[6] I love using `data.table` for this. Check out https://www.ashwinmalshe.com/post/speed-comparison-rbind/

Looks good.

7.5 Fama-French factors

Fama-French factors are available from Kenneth French's website. We will use Fama-French 5 factors daily file. More description on what these factors are is available here.

I earlier downloaded and cleaned up the CSV file a little bit for easy operation. You can download it from this link: http://bit.ly/2V5euiB or you can directly read it as follows.

```
ff <- read.csv("http://bit.ly/2V5euiB") %>%
  mutate(Date = lubridate::ymd(Date)) %>%
  mutate_if(is.numeric, function(x) x / 100)
```

We divided all the returns by 100 to correctly convert them into fractions. Take a look at the data set.

```
head(ff)
```

```
##         Date    MktRF     SMB     HML     RMW     CMA      RF
## 1 1963-07-01 -0.0067  0.0000 -0.0032 -0.0001  0.0015 0.00012
## 2 1963-07-02  0.0079 -0.0027  0.0027 -0.0007 -0.0019 0.00012
## 3 1963-07-03  0.0063 -0.0017 -0.0009  0.0017 -0.0033 0.00012
## 4 1963-07-05  0.0040  0.0008 -0.0028  0.0008 -0.0033 0.00012
## 5 1963-07-08 -0.0063  0.0004 -0.0018 -0.0029  0.0013 0.00012
## 6 1963-07-09  0.0045  0.0000  0.0010  0.0014 -0.0004 0.00012
```

Next, we will merge Fama-French factors with `price_stacked`.

```
price_merge <- price_stakced %>%
  inner_join(ff, by = c("ref.date" = "Date")) %>%
  mutate(ret = ret.adjusted.prices - RF)
```

`ret` is $ExcessRet_{it}$

7.6 Parameter estimates

We will first use linear regression to obtain parameter estimates. In the following code we will run regression model separately for each company and save them in a large list called `lm_list`

```
lm_list <- price_merge %>%
  filter(-260 <= event_diff & event_diff < -10 ) %>%
  split(.$ticker) %>%
  purrr::map(~ lm(ret ~ MktRF + SMB + HML + RMW + CMA, data = .))
```

Note that we could have simply extracted the coefficients from each linear regression to calculate predicted returns manually. However, we will need the prediction intervals and base R's `predict()` function produces them automatically.

7.7 Predict stock return on the event day

First, keep only the event days.

```
pred_dt <- price_merge %>%
  filter(event_diff == 0)
```

Next, make the predictions and store them in a dataframe called `predictions`. Note that `predict()` function, when used with the argument `interval` will output 3 values labeled as `fit`, `lwr`, and `uhr`. The last two are the lower and upper prediction intervals, respectively.

```
predictions <- structure(list(fit = numeric(),
                              lwr = numeric(),
                              upr = numeric()),
                         class = "data.frame")

for (i in 1:length(lm_list)) {
  predictions <- rbind(predictions,
                       predict(lm_list[[i]],
                               pred_dt[i, ],
                               interval = "predict")
                       )
}
```

Finally, add the columns for real returns (adjusted for the risk-free returns) and ticker.

```
predictions <- predictions %>%
  mutate(ticker = pred_dt$ticker,
         ret = pred_dt$ret)
```

7.8 Plotting the returns

We will now plot the returns for easy comparison. The best way to depict these returns will be by using a scatterplot superimposed on the error bars corresponding to the lower and upper prediction intervals.

We will first create a data set that is reshaped to be long. As the realized returns do not have the prediction intervals, we will drop these variables for the time being.

Table 7.2: Reshaped Predictions

	ticker	variable	value
1	AMZN	ret	-0.05
2	BA	ret	0
3	CBS	ret	0
4	CCZ	ret	0
...	NA	NA	...
19	MKGAF	fit	0
20	NYT	fit	0
21	JWN	fit	0
22	TM	fit	-0.01

```
predictions_lg <- predictions %>%
  select(ticker, ret, fit) %>%
  reshape2::melt(id.vars = "ticker")
```

Take a look at this data using `headTail()` function from `psych` package, which will print 4 observations from the top and 4 observations from the bottom by default.

```
psych::headTail(predictions_lg) %>%
  knitr::kable(caption = "Reshaped Predictions",
               booktabs = TRUE)
```

Now make the plot.

```
ggplot(predictions_lg, aes(x = ticker)) +
  geom_errorbar(aes(ymin = lwr, ymax = upr),
            data = predictions,
            color = "#03c3f6",
            width = 0.2) +
  geom_point(aes(y = value,
                 fill = variable),
             color = "#3b4252",
             shape = 21) +
  scale_y_continuous(labels = scales::percent) +
  scale_fill_manual(values = c("#ef2e69",
                               "#205aff"),
                    labels = c("Realized Returns",
                               "Estimated Returns")) +
  labs(x = "Ticker",
       y = "Daily Stock Returns",
       fill = "") +
  theme_minimal()
```

Interestingly, except for Amazon, no other stock suffered from Trump attack! Nordstrom actually showed an unexpected increase in the stock price. Otherwise, rest 9 stocks have no effect of

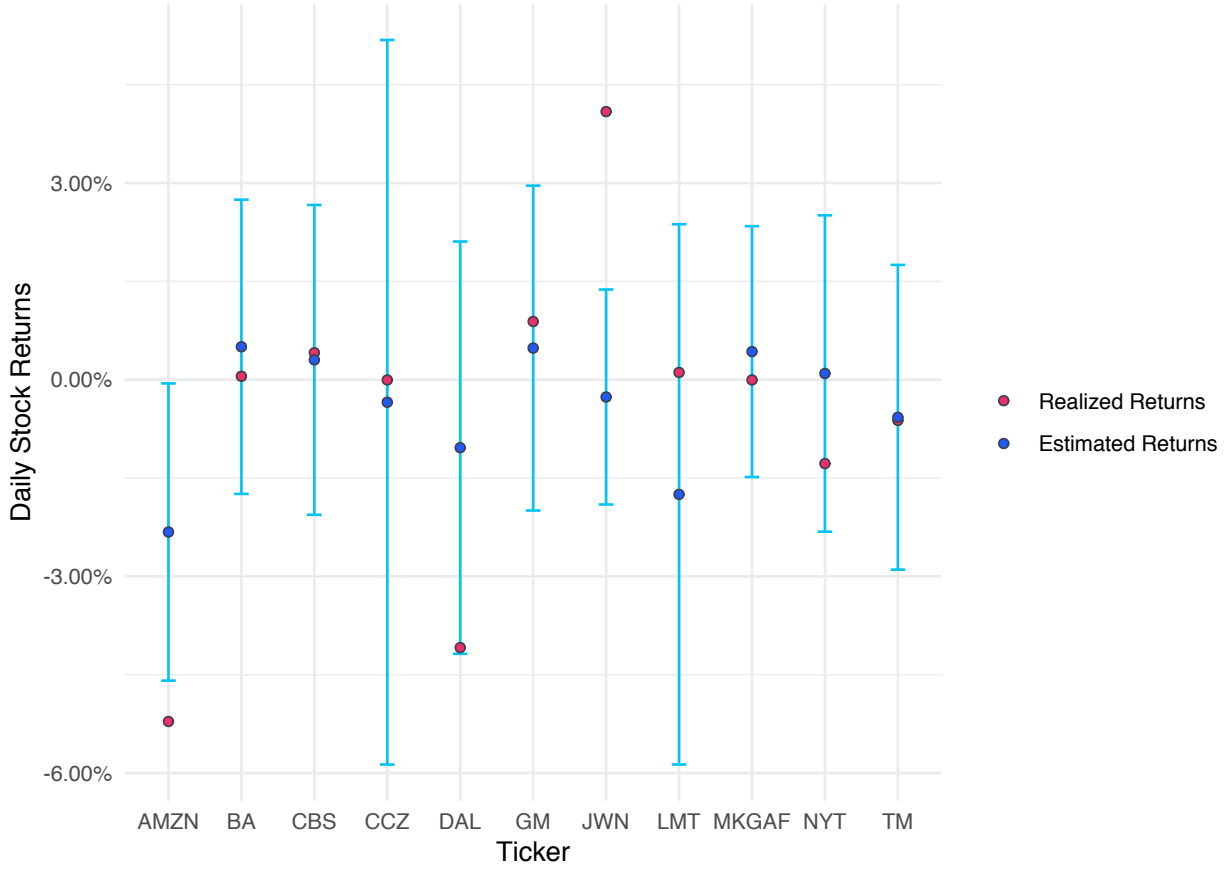


Figure 7.2: Stock Reactions to Trump Attacks

Trump Twitter attack.

7.9 FF model efficacy

On one hand it is good to know that Trump's attacks are not hurting shareholder value. On the other hand, we might be wrong to conclude this because our analysis totally relies on the efficacy of the FF model as a solid benchmark. Given the large prediction intervals above, I am not sure the model has done a good job of capturing the real risk measures! They look very noisy to me.

We can quickly check the R^2 of the regression models to figure out how well FF model described the stock prices over a year.

```
price_merge %>%
  filter(-260 <= event_diff & event_diff < -10 ) %>%
  split(.$ticker) %>%
  map(~ lm(ret ~ MktRF + SMB + HML + RMW + CMA, data = .)) %>%
  map(summary) %>%
  map_dbl("r.squared")
```

```
##       AMZN         BA        CBS        CCZ        DAL         GM
## 0.43557083 0.49741467 0.28645931 0.01666242 0.36784969 0.41454235
##        JWN        LMT       MKGAF        NYT         TM
## 0.29005370 0.19749228 0.01666749 0.52996719 0.41700821
```

Looks like the R^2 range from just 1.67% (CCZ and MKGAF) to 53% (NYSE). So, at least in some cases our benchmark model was not very good. Perhaps we can improve the model by using nonlinear transformations of the variables. You could also try other machine learning models such as random Forest and Support Vector Machine. However, a major issue while doing cross-validation for these models on the training data is that you will lose the time-series aspect. Although we are not using the time-series property of data in the linear model either, we are strictly forecasting in future. With cross-validation, the model can be predicting past values based on the future values! So even when the model may perform well in sample, it might be meaningless for predicting future values.

7.10 Summary

This exercise showed you how to do event study. Although this was a basic example, it conveys the importance of event study methodology. The critical aspect of an event study is that we are required to have a benchmark, which typically is a factor model such as Fama-French 3 or 5 factor model. Whether this model is the best benchmark is certainly up for a debate.

Using event study methodology we showed that Donald Trump's Twitter attacks on brands are not necessarily leading up to shareholder value loss. Although we found that Amazon shareholders were harmed by Trump tweet, other brands did not have any negative impact.

Chapter 8

Cluster Analysis

```
Most of the code in this book chapter is thanks to Pallav Routh who is a
PhD student in my department.
```

Imagine that a marketer wants to group customers into identifiable groups. The marketer has customer demographic as well as purchase behavior data on these customers. The objective then is to create groups such that the customers within each group are homogeneous and customers in any two groups are heterogeneous. As these groups are not yet formed, there is no "target" variable that the marketer can use to build a predictive model. All the marketer has is the data on customer characteristics and purchase behavior. In machine learning, a modeling problem without a labelled target variable is called unsupervised learning problem. In this specific case of customer segmentation, cluster analysis turns out to be a highly popular unsupervised learning method.

Cluster analysis or clustering is a task that can be completed using many different algorithms. In this exercise, we will use *k-means* clustering, which identifies k number of clusters (or groups) that gives us the low variability within a cluster and high variability between two clusters.

8.1 Tasks to complete

1. Using customer purchase behavior data perform cluster analysis to determine the optimum number of customer segments.
2. Visualize customer segments.
3. Describe the customer segments to help marketers target customers.

8.2 Data description

We will use two **simulated** data sets from `SMCRM` package. This package consists of 7 data sets used in the book titled "Statistical Methods in Customer Relationship Management"

Table 8.1: Variable Description for 'customerRetentionTransactions'

Variable	Description
customer	customer number (from 1 to 500)
quarter	quarter (from 1 to 12) where the transactions occurred
purchase	1 when the customer purchased in the given quarter and 0 if no purchase occurred in that quarter
order_quantity	dollar value of the purchases in the given quarter
crossbuy	number of different categories purchased in a given quarter
ret_expense	dollars spent on marketing efforts to try and retain that customer in the given quarter
ret_expense_sq	square of dollars spent on marketing efforts to try and retain that customer in the given quarter

Table 8.2: Variable Description for 'customerRetentionDemographics'

Variable	Description
customer	customer number (from 1 to 500)
gender	1 if the customer is male, 0 if the customer is female
married	1 if the customer is married, 0 if the customer is not married
income	1 if income <= $30,000; 2 if $30,000 < income <= $45,000; 3 if $45,000 < income <= $60,000; 4 if $60,000 < income <= $75,000; 5 if $75,000 < income <= $90,000; 6 if income > $90,000
first_purchase	value of the first purchase made by the customer in quarter 1
loyalty	1 if the customer is a member of the loyalty program, 0 if not
sow	share-of-wallet; the percentage of purchases the customer makes from the given firm given the total amount of purchases across all firms in that category
clv	discounted value of all expected future profits, or customer lifetime value

by Kumar and Petersen. Specifically we will use `customerRetentionTransactions` and `customerRetentionDemographics` data sets. The first data consists of customer transactions information on 500 customers over 12 quarters. There are no missing values so every customer has 12 observations and accordingly there are 6,000 observations (500*12). The second data set contains demographic information on 500 customers. This information is time-invariant.

The data sets and their descriptions are as follows:

8.2.1 `customerRetentionTransactions`

8.2.2 `customerRetentionDemographics`

8.3 Packages and data

Load the required packages for this exercise. All the packages are available on CRAN if you need to install them.

```
library(dplyr)
library(SMCRM) # For the data sets
library(factoextra) # Cluster visualization
library(plotly) # Interactive plotting
library(caret)
library(plyr)
```

8.3.1 Data

The following code will load the data sets in your global environment. You can rename them once loaded.

```
data("customerRetentionTransactions")
data("customerRetentionDemographics")
```

8.4 Clustering variables

In marketing, three variables from customer purchase history are known to play a bit role in predicting future purchases. These variables are recency, frequency, and monetary value. Analysis involving these variables is called RFM analysis. In our context, these are measured as:

1. Recency: The number of quarters since the last purchase
2. Frequency: The number of times transacted over 12 quarters. As the exact number of transactions is unavailable, we assume that customer has transacted once in a quarter, if the total expenditure is non-zero.
3. Monetary value: The total dollar value of transaction in 12 quarters.

Accordingly, we create a new data set with these variables. The last two variables are straightforward to calculate so we will first generate a data set with these variables.

```
cluster_data1 <-
  customerRetentionTransactions %>%
    group_by(customer) %>%
     summarize(frequency = sum(purchase),
               monetary_value = sum(order_quantity)) %>%
    ungroup()
```

Next, we create a column for recency and save it in another data frame.

```
cluster_data2 <-
  customerRetentionTransactions %>%
               filter(purchase == 1) %>%
               group_by(customer) %>%
                  summarise(last_transaction = last(quarter)) %>%
                  mutate(recency = 12 - last_transaction) %>%
               ungroup() %>%
               select(-last_transaction)
```

Finally, we will merge these two data sets. We will also merge the demographics data.

```
cluster_data <- inner_join(cluster_data1,
                           cluster_data2,
                           by = "customer") %>%
  inner_join(customerRetentionDemographics,
             by = "customer")
```

8.5 Scaling variables

Cluster analysis can use any numeric variable for clustering. The algorithms rely on a distance metric and therefore it is preferable to scale all the variables to map on to the same scale. The easiest scaling is converting data into z scores which involves mean centering a variable and then scaling that variable by the standard deviation. The `scale()` function from base R performs centering and scaling in one step.

```
cluster_data_pro <- cluster_data %>%
  select(recency, frequency, monetary_value) %>%
  scale() %>%
  as.data.frame()
```

Verify that we have 0 means and 1 standard deviations

```
print("Means")
```

```
## [1] "Means"
```

```
sapply(cluster_data_pro, mean) %>%  round(4)
```

```
##        recency      frequency monetary_value
##              0              0              0
```

```
print("Standard Deviations")
```

```
## [1] "Standard Deviations"
```

```
sapply(cluster_data_pro, sd)
```

```
##        recency      frequency monetary_value
##              1              1              1
```

8.6 K-means clustering

K-means clustering is a hungry algorithm. It will randomly pick up some points at the beginning as the cluster centroids and then keep updating clusters depending on the distance metric.

8.7 Cluster feasibility

Before applying the clustering algorithm, it is best to assess if the data will yield any meaningful clusters. One popular method is Hopkin's statistic that tests whether data has uniform distribution. Clustering is only possible when the distribution shows heterogeneous regions. Hopkin's statistic ranges between 0 and 1. Depending on the way the formula is implemented in the package, clusterability of the data is determined by how close Hopkin's statistic is to 0 or 1. We will use `get_clust_tendency()` from `factoextra` package. According to this function the closer Hopkin's statistic is to 0, the better is the clusterability in the data.

This function also output a plot showing the clusters. The plot looks at the dissimilarity matrix by computing the distance between points. We ideally want to see blocks of similar color.

```
set.seed(4569)

cluster_td <-
  get_clust_tendency(
    cluster_data_pro,
    n = 400, # Pick 400 points randomly
    gradient = list(low = "steelblue",
                    high = "white")
    )
```

Print Hopkin's statistic

```
cluster_td$hopkins_stat
```

```
## [1] 0.03101145
```

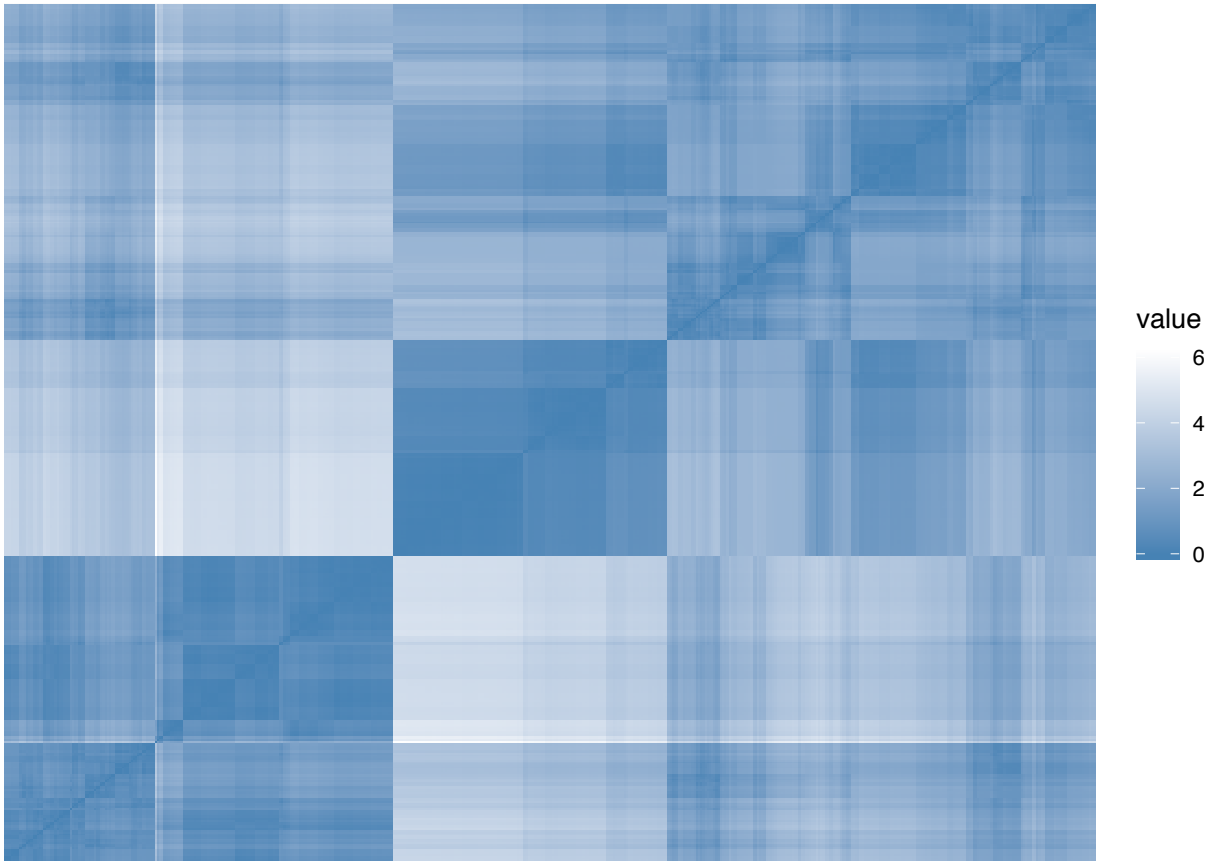


Figure 8.1: Cluster Feasibility Plot

Hopkin's stat is close to zero, which suggests that data is clusterable. Let's take a look at Figure 8.1, which also supports that there are blocks of data that can be clustered.

```
cluster_td$plot
```

8.8 Optimal number of clusters

Ideally the cluster analysis should result in clusters with low within cluster variance and high between cluster variance. The first "elbow" on the plot gives us the optimal number of clusters. We will use `fviz_nbclust()` function from `factoextra` package. In argument `wss` stands for "within sum of squares". As we increase the number of clusters, `wss` should go down. Usually in the plot we see a kink or elbow after which `wss` flattens out. This is a somewhat subjective process.

```
fviz_nbclust(cluster_data_pro, kmeans, method = "wss")
```

In Figure 8.2, the kink occurs when the number of cluster is at 4 or 5 depending how sensitive you are to the decrease in `wss` visually. For this example, we will consider 4 clusters as optimum.

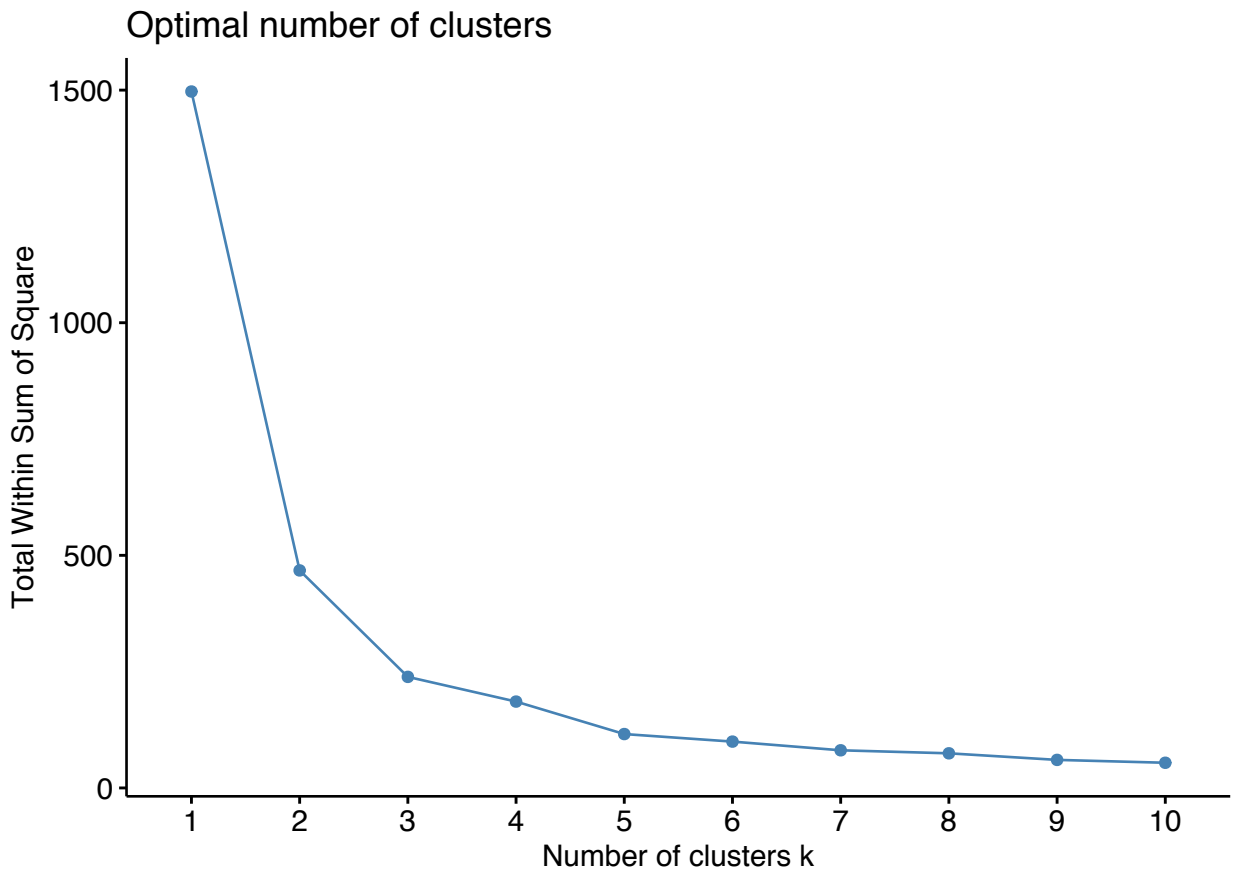


Figure 8.2: Optimal Number of Clusters

8.9 k-means to create clusters

Now we are ready to perform clustering. As we are using k-means clustering, we will use `kmeans()` function from base R's `stats` package. Recall that k-means is a hungry algorithm and it starts making clusters randomly. In order to avoid a situation where we start off on sub optimal points, `kmeans()` allows us to specify a number of starting points to try. The `nstart` argument in the function specifies this number.

```
set.seed(309)

km_cluster <- kmeans(cluster_data_pro,
                     4,
                     nstart = 25)
```

`km_cluster` is a list of 9 items. we are interested in the cluster membership of each point, which is stored in `km_cluster$cluster`. Add these clusters to the data.

```
cluster_data$km_cluster <- km_cluster$cluster
```

8.10 Visualize clusters

The `fviz_cluster()` function visualizes the cluster in 2 dimensions. However, we have 3 dimensions. `fviz_cluster()` performs Principle Components Analysis (PCA)[1] behind the scenes to reduce the dimensions such that data can be represented by clusters in a 2-D space.

```
fviz_cluster(object = km_cluster, # kmeans object
             data = cluster_data_pro, # data used for clustering
             ellipse.type = "norm",
             geom = "point",
             palette = "jco",
             main = "",
             ggtheme = theme_minimal())
```

Figure 8.3 shows that we have really neat clusters with not a lot of overlap.

Although, not possible in all the cases, for our exercise we can also build a 3-D plot because we have only 3 clustering variables. We will use `plotly` package for making an interactive visualization. The code below, without explanation, shows you how to make an interactive 3-D plot. Interactivity will not work in PDF. The HTML plot is shown in Figure 8.4.

```
plot_ly(x = cluster_data$recency,
        y = cluster_data$frequency,
```

[1] PCA is a dimensionality reduction technique.

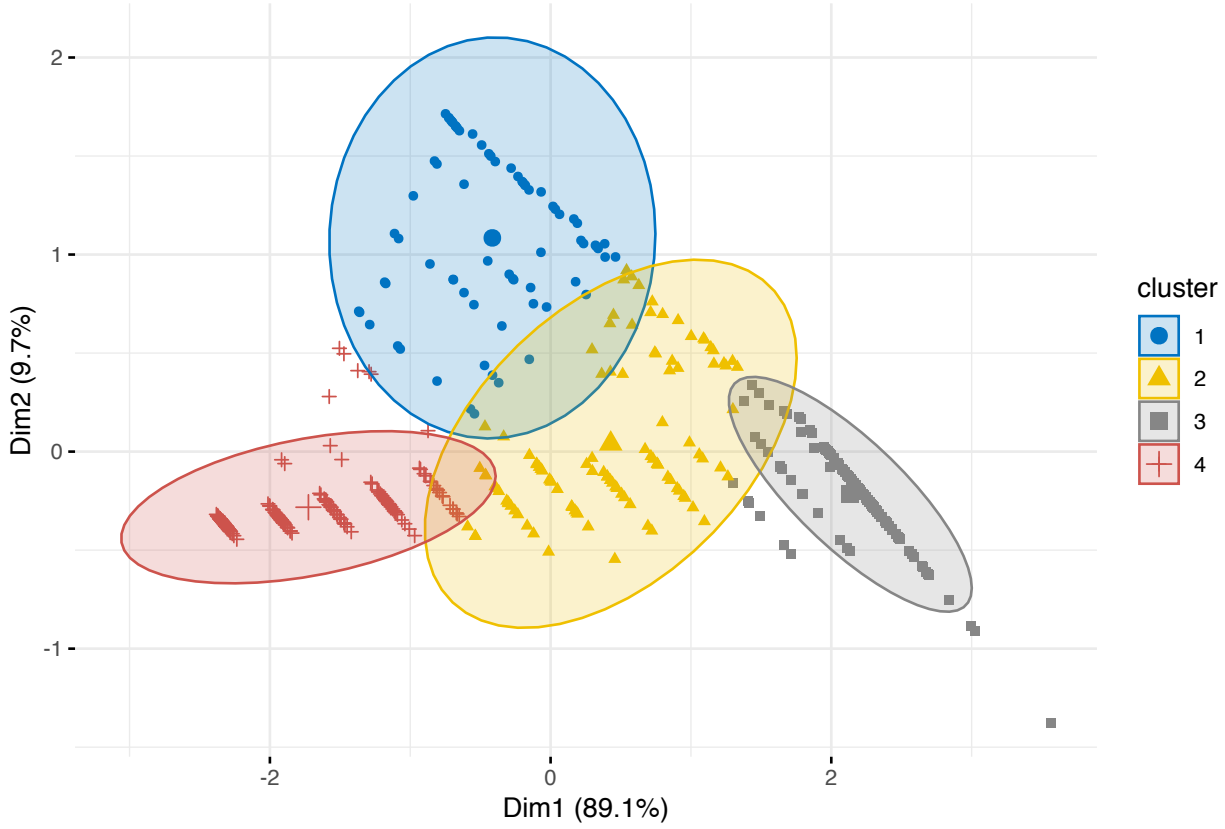


Figure 8.3: Cluster Plot

Figure 8.4: 3-D Clusters

```
        z = cluster_data$monetary_value,
        type = "scatter3d",
        mode = "markers",
        color = as.factor(cluster_data$km_cluster)) %>%
        layout(title = "",
               scene = list(xaxis = list(title = "Recency"),
                            yaxis = list(title = "Frequency"),
                            zaxis = list(title = "Monetary value")))
```

8.11 Cluster characteristics

Once we form the clusters, our next task is to find the characteristics of the customers in those clusters. First, we will see the average values pertaining to recency, frequency, and monetary value.

Table 8.3: Cluster Characteristics

km_cluster	mean_recency	mean_frequency	mean_mv	num_members
1	1.514	3.931	688.823	72
2	3.298	7.567	1859.503	104
3	0.191	11.702	3368.381	141
4	9.350	2.503	513.232	183

```
cluster_data %>%
  group_by(km_cluster) %>%
  summarise(mean_recency = mean(recency),
            mean_frequency = mean(frequency),
            mean_mv = mean(monetary_value),
            members = n()) %>%
  mutate_all(round, 3)
```

Table 8.3 shows that customers in cluster 1 spent $3,368 in the last 12 quarters and have low recency and high frequency.[2] Cluster 2 customers seem to be the worst lot because they spent only $513 in the last 12 quarters with high recency and low frequency. It's possible that the customers in the segment are no longer planning to return and buy anything. Thus, the firm may have lost them.

```
An important exercise at this point is to name these customer segments.
For example, cluster 2 can be named "Lost Cause" because they are probably
not returning. Can you think of names for other customer segments?
```

8.12 Explore the segments

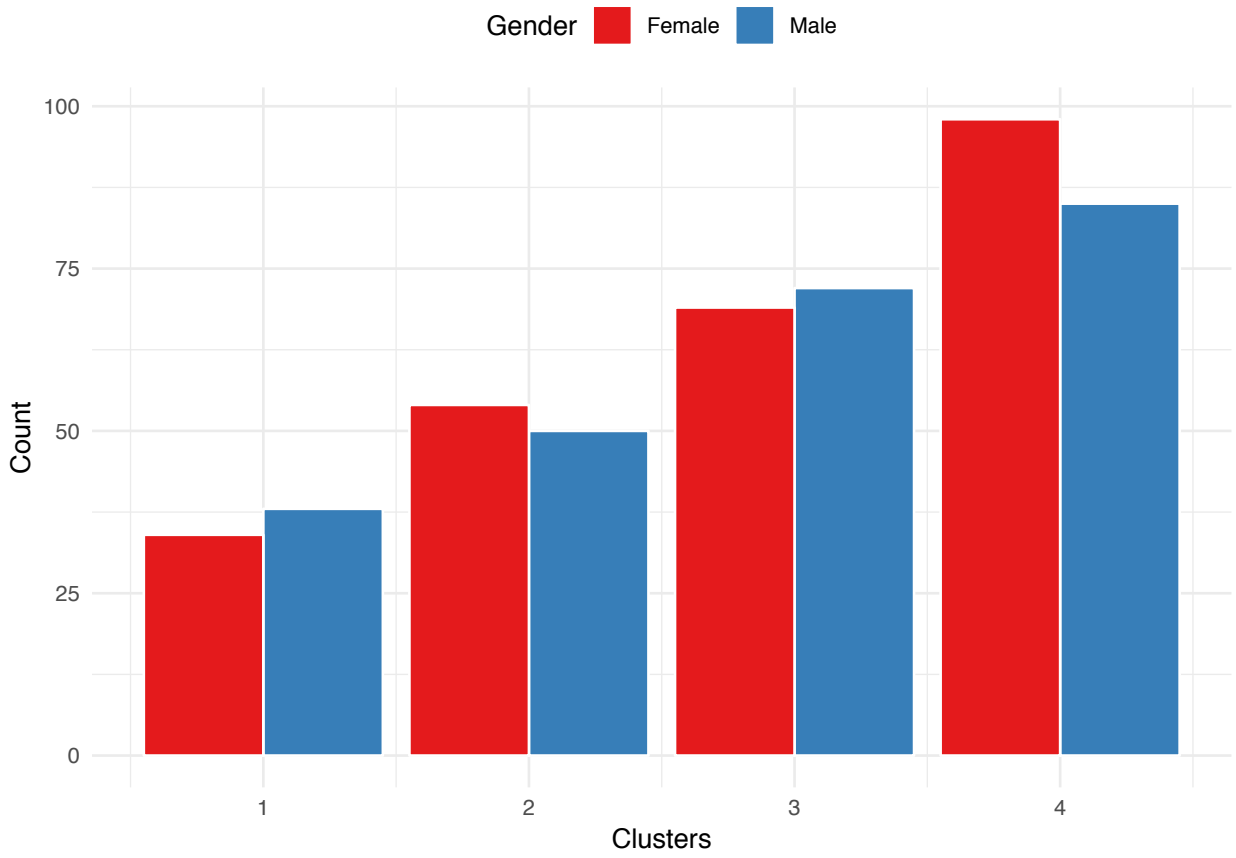
Now that we have isolated 4 customer segments from our existing customers, we shift our attention to a practical problem. If marketers want to use this information to target *new* customers, they will need description of the customers that they can use to locate these customers. By definition, a new customer has not purchased from you before. Therefore, you don't have their purchase behavior data. However, if we can correlate customer demographics with the customer segments, then we can help marketers in identifying these segments.

Let's investigate 3 demographic variables —gender, marital status, and income— within each cluster.

8.12.1 Gender

[2] Low recency means that these customers made purchases very recently.

```
cluster_data %>%
  mutate(Gender = ifelse(gender == 1, "Male", "Female")) %>%
  group_by(km_cluster, Gender) %>%
  summarise(count = n()) %>%
  ggplot(aes(x = km_cluster, y = count, fill = Gender)) +
   geom_col(position = "dodge", color = "white") +
   scale_fill_brewer(palette = "Set1") +
   labs(y = "Count", x = "Clusters") +
   theme_minimal() +
   theme(legend.position = "top")
```

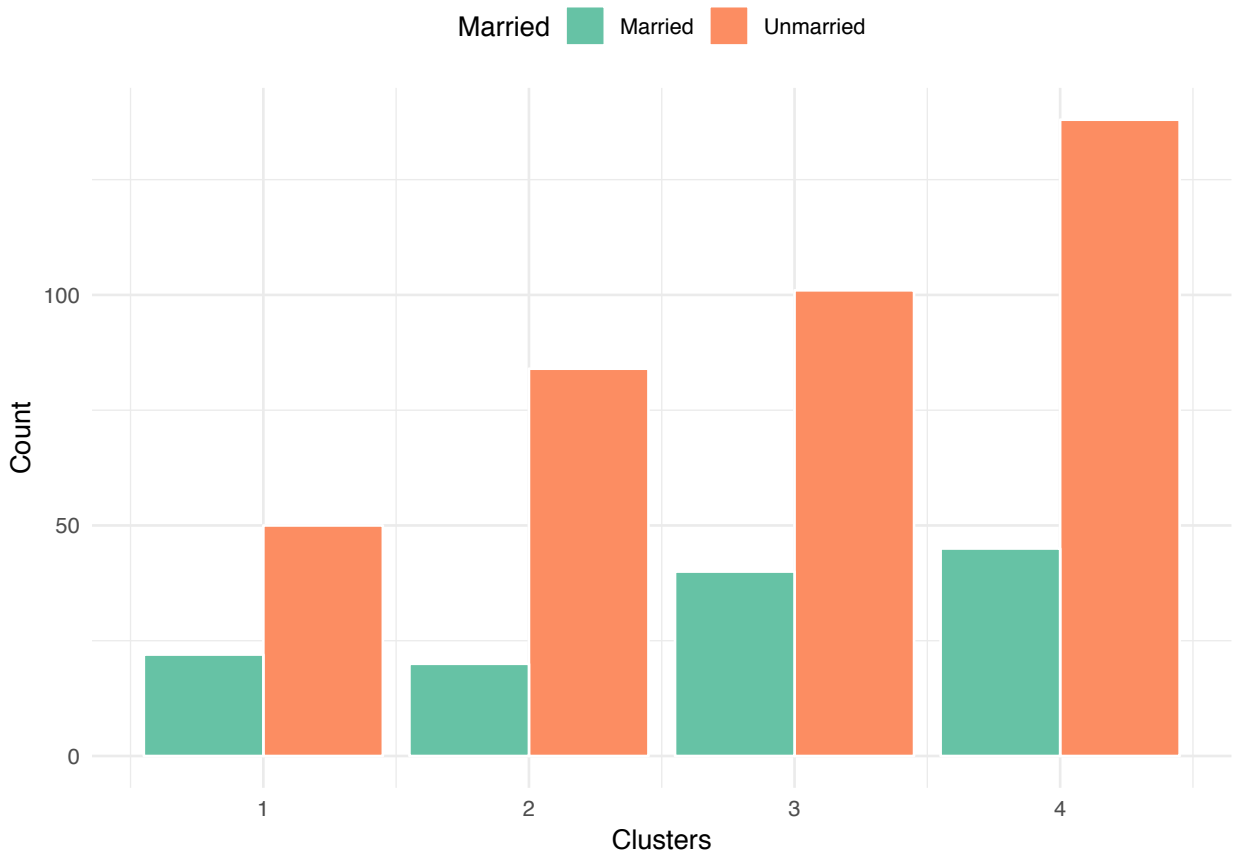


Cluster 1, 3, and 4 have even distributions of males and females. Cluster 2 has a slightly higher concentration of females. However, targeting based on gender alone may not give significant sales.

8.12.2 Marital status

```
cluster_data %>%
  mutate(Married = ifelse(married == 1,
```

```
                        "Married",
                        "Unmarried")) %>%
  group_by(km_cluster, Married) %>%
  summarise(count = n()) %>%
  ggplot(aes(x = km_cluster, y = count, fill = Married)) +
   geom_col(position = "dodge", color = "white") +
   scale_fill_brewer(palette = "Set2") +
   labs(y = "Count", x = "Clusters") +
   theme_minimal() +
   theme(legend.position = "top")
```

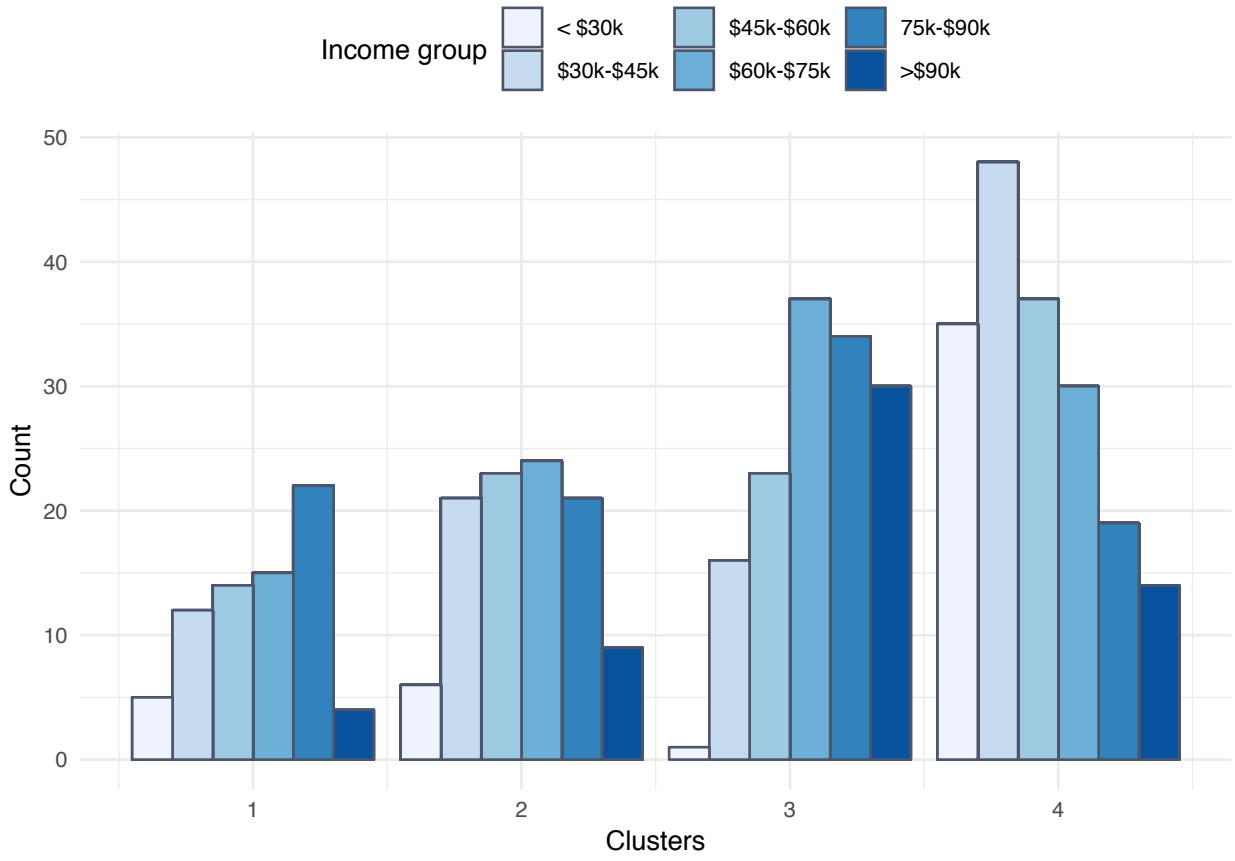


In general, all clusters have higher concentration of unmarried customers compared to married. Thus, marital status is not a good discriminant variable.

8.12.3 Income

```
cluster_data %>%
  mutate(Income = plyr::mapvalues(
    income,
```

```
    from = c(1:6),
    to = c("< $30k", "$30k-$45k","$45k-$60k","$60k-$75k",
           "75k-$90k",">$90k"))) %>%
  group_by(km_cluster, Income) %>%
   mutate(count = n()) %>%
   ggplot(aes(x = km_cluster,
              y = count,
              fill = reorder(Income, income))) +
   geom_col(position = "dodge", color = "#4c566a") +
   scale_fill_brewer("Income group") +
   labs(y = "Count", x = "Clusters") +
   theme_minimal() +
   theme(legend.position = "top")
```



Clearly, `income` varies a lot between these 4 segments.

8.13 Random forest for segment description

Finally, we will use random forest for creating segment definitions. If you want to read more on building a predictive model using random forest, please see Section 4.3.

8.13.1 Prepare the data

We will make minor changes to the data set. First, we will drop irrelevant variables. Next we will recode `gender`, `married`, `income`, and `km_cluster` so that their levels are character variables rather than numbers. We then reclassify these 4 variables as factors. This is essential because R will internally create dummy variables for factors. The factor levels will be used to create internal variable names, which will work only for character values of the levels.

```
cluster_dt_rf <- cluster_data %>%
  select(-c(customer, frequency, monetary_value, recency)) %>%
  mutate(gender = plyr::mapvalues(
                  gender,
                  from = c(0, 1),
                  to = c("f", "m")),
         married = plyr::mapvalues(married,
                   from = c(0,1 ),
                   to = c("no", "yes")),
         income = plyr::mapvalues(income,
                  from = c(1:6),
                  to = c("i1", "i2", "i3",
                         "i4", "i5", "i6")),
         km_cluster = plyr::mapvalues(km_cluster,
                      from = c(1:4),
                      to = c("c1", "c2", "c3", "c4"))) %>%
  mutate_at(vars(gender, married, income, km_cluster), as.factor)
```

8.13.2 Create train and test data sets

```
index <- createDataPartition(cluster_dt_rf$km_cluster,
                             p = 0.8,
                             list = FALSE)
```

```
train_dt <- cluster_dt_rf[index, ]
test_dt <- cluster_dt_rf[-index, ]
```

8.13.3 Set up train control

We are using 10-fold cross-validation.

```
trControl <- trainControl(method = "cv", #crossvalidation
                          number = 10,   # 10 folds
                          search = "grid",
```

```
                          classProbs = TRUE  #computes class probabilities
                          )

tuneGrid_large <- expand.grid(mtry = c(1:(ncol(train_dt) - 2)))
```

8.13.4 Train the model. This will take a few minutes.

```
set.seed(2222)

modelRF_large <- train(km_cluster ~ . ,
                  data = train_dt,
                  method = "rf",
                  metric = "Accuracy",
                  tuneGrid = tuneGrid_large,
                  trControl = trControl,
                  ntree = 1000)
```

Check which `mtry` gives us the best result.

```
print(modelRF_large)
```

```
## Random Forest
##
## 402 samples
##   7 predictor
##   4 classes: 'c1', 'c2', 'c3', 'c4'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 362, 363, 361, 361, 362, 363, ...
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa
##   1     0.6138886  0.4144991
##   2     0.7857429  0.6930965
##   3     0.7855629  0.6944846
##   4     0.7779925  0.6853990
##   5     0.7830597  0.6925895
##   6     0.7806207  0.6891800
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 2.
```

Turns out that the model with `mtry = 2` is the best.

8.13.5 Get the variable importance

```
varImp(modelRF_large, scale = TRUE)
```

```
## rf variable importance
##
##                 Overall
## clv            100.0000
## sow             74.0256
## first_purchase  20.1671
## incomei5         4.0889
## genderm          1.7341
## loyalty          1.4986
## marriedyes       1.4867
## incomei6         0.6776
## incomei2         0.6415
## incomei4         0.3877
## incomei3         0.0000
```

`clv` and `sow` are the most important predictors. The cluster analysis effectively clustered people based on their customer lifetime value and share of wallet![3]

8.14 Model performance

Finally, let's assess the model performance by using it on the test data set.

```
confusionMatrix(predict(modelRF_large,
                        select(test_dt, -km_cluster)),
                reference = test_dt$km_cluster,
                positive = "Yes")
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction c1 c2 c3 c4
##         c1  0  1 27  0
##         c2  6  0  0 35
##         c3  8  0  0  0
##         c4  0 19  1  1
##
## Overall Statistics
```

[3] If this is going to be used to identify new customers, where do we get data on CLV and SOW?

```
##
##                Accuracy : 0.0102
##                  95% CI : (3e-04, 0.0555)
##     No Information Rate : 0.3673
##     P-Value [Acc > NIR] : 1
##
##                   Kappa : -0.2825
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: c1 Class: c2 Class: c3 Class: c4
## Sensitivity             0.0000    0.0000   0.00000   0.02778
## Specificity             0.6667    0.4744   0.88571   0.67742
## Pos Pred Value          0.0000    0.0000   0.00000   0.04762
## Neg Pred Value          0.8000    0.6491   0.68889   0.54545
## Prevalence              0.1429    0.2041   0.28571   0.36735
## Detection Rate          0.0000    0.0000   0.00000   0.01020
## Detection Prevalence    0.2857    0.4184   0.08163   0.21429
## Balanced Accuracy       0.3333    0.2372   0.44286   0.35260
```

The model does a fairly good job of predicting the customer segments out of sample. We get almost 78% accuracy and Kappa equaling 0.68. The model over-classified customers in segment 2. All the misclassifications are predominantly because of classifying segment 3 customers as segment 3. Perhaps these two segments are quite close to each other on the predictor variables. On the other hand segment 1 and 4 classifications are quite accurate.

8.15 Summary

In this exercise, we segmented customers in 4 groups based on their past purchase behavior. We used recency, frequency, and monetary value (RFM) to segment customers. K-means clustering led to 4 segments. Next we described the segments using customer demographics. Using random forest classifier, we found out that customer lifetime value and share of wallet are two critical metrics for segmenting customers.

```
library(knitr)
library(recommenderlab)
library(ggplot2)
library(dplyr)
library(here)
```

Chapter 9

Collaborative Filtering

Collaborative filtering (CF) is a method used in building recommender systems on big data. Common applications include Amazon product recommendations, Netflix movies and shows recommendations, iTunes music recommendations, etc. Figures 9.1, 9.2, and 9.3 show examples of recommendations for me from these 3 services.

CF uses a user-item rating matrix, which contains rating given by users to items. Beyond this, CF does not require other user information such as demographics (e.g., age, sex, etc.) or item information (e.g., movie genre, type of music, etc.). This is a key strength of CF because it relies on minimal information and doesn't raise major privacy concerns as it does not need to know any personal information about the user outside their previous behavior. CF doesn't rely on the content analysis of the items, which makes it easy to build recommenders for applications which may otherwise require complex machine learning models. For instance, CF can be used to recommend jokes without actually knowing what those jokes are!

As the name suggests, the "collaborative" part means that the method relies on behavior by other users with similar tastes. The user-item rating matrix does not explicitly show which users are similar or which items are similar. The objective of CF is then to identify these similarities.

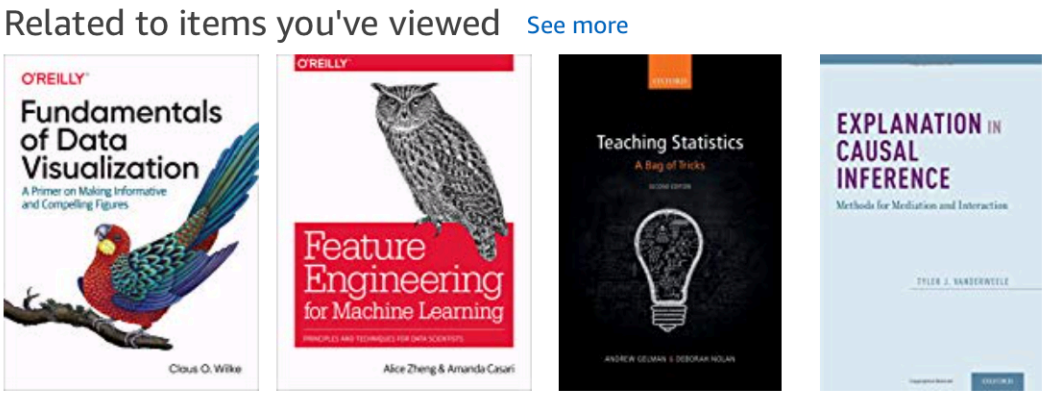


Figure 9.1: Amazon Recommendations

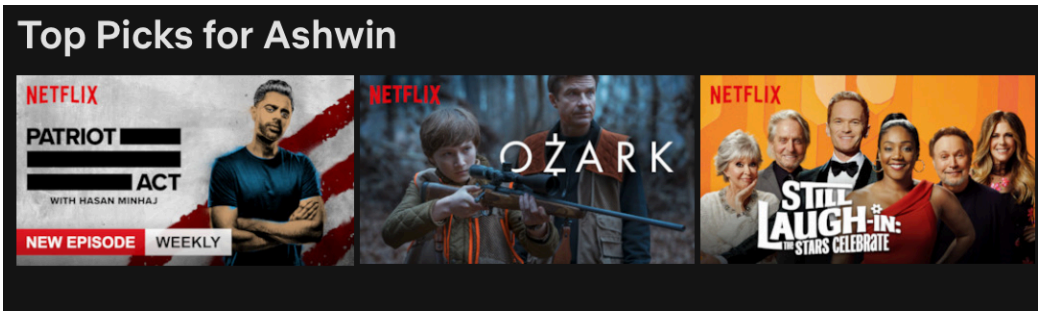


Figure 9.2: Netflix Recommendations

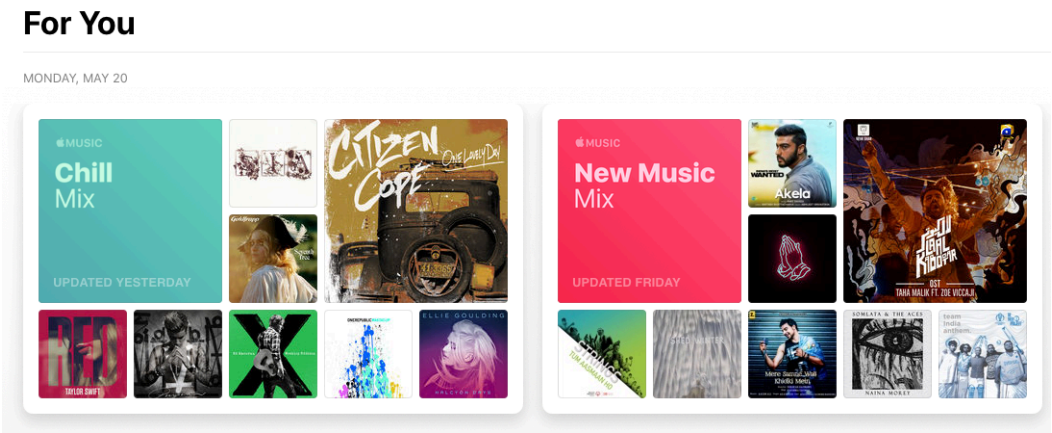


Figure 9.3: iTunes Recommendations

9.0.1 An example

Imagine that you and your friend A decide to meet for lunch. Two of you go to a restaurant where you have not eaten before. But A visits this place often because it's close to her office. You are seated at a table and the server leaves you with menus. You are unsure about what to order so you turn to your friend for suggestions. A recommends a new vegetarian burger called "Beyond burger". How likely are you to try this burger if...

1. You and your friend A have very similar preferences for food
2. You and your friend A have not so similar preferences for food

I guess that you are more likely to order Beyond burger in the first case, that is, when you and A have very similar food preferences. This is in essence what CF entails. However, rather than recommending an item based on only 1 other user's experience, CF uses a lot of users to make more reliable prediction. Think about it as an app that has food preferences of all your friends. This app can recommend you food items by taking into account the evaluations of all the friends with similar taste as you.

9.1 Types of collaborative filtering

In this exercise, we will see many types of CF methods. I briefly describe this methods below.

1. User-based collaborative filtering (UBCF): UBCF relies on the assumption that users with similar preferences in the past will have similar preferences in the future. Thus, if we know that two users have given very similar ratings to a set of movies, we can use the movie rating from one user to predict the rating from the second user on a yet unseen movie.
2. Item-based collaborative filtering (IBCF): IBCF relies on the assumption that a user will prefer an item which is similar to the items they have previously liked. For instance, if you like comedies, IBCF will recommend you more comedies in future.
3. Popularity based collaborative filtering: In this case CF simply recommends a list of popular items. These recommendations are not tailored to any specific user.
4. Singular Value Decomposition based collaborative filtering (SVD): CF using SVD are not exactly using SVD but instead they are using a method that is quite similar to SVD. SVD is a matrix factorization method, which can be used for dimensionality reduction. SVD based CF identifies latent factors that group similar users and similar items. This reduces the burden on recommender dealing with really large amounts of data. Consider that Netflix has hundreds of millions of users and thousands if not millions of movie and show titles.

In 2006, Netflix ran a recommender contest with $1 million first prize. The contest led to a lot of enhancement in collaborative filtering methods. One such enhancement is Funk SVD due to Simon Funk.[1] You can read a simplified explanation on this Medium blog. Luckily `recommenderlab` implements Funk SVD as well.

[1] https://sifter.org/simon/journal/20061211.html

9.2 A note on sparsity of rating matrix

The user-item rating matrix is a sparse matrix because most people do not provide ratings. Think about how many times you provided any ratings on Amazon or Netflix. Therefore, most cells of a rating matrix have missing values.

A way to improve upon this situation is to use 1-0 rating scheme whereby if a user engages with an item (buys a book, watches a movie, listens to a song), we fill the cell with 1 and otherwise it is filled with 0. This still does not solve the problem because a 0 has more than one meaning. You may not watch a movie because you don't like something about the movie (genre, actors) or you are simply not aware that the movie exists.

We will side-step this concern for the current exercise. In reality, you will have to find ways to address this because your recommender efficacy depends on it. Ideally you would like to recommend an item that the user is likely to appreciate and engage with.

9.3 `recommenderlab` package

We will use `recommenderlab` package to build recommender systems. This package is designed for testing recommender systems in the lab setting rather than in the production setting. The package is available on CRAN.

```
library(recommenderlab)
library(ggplot2)
library(dplyr)
```

9.3.1 MovieLense data

We will use MovieLense data, which is bundled with `recommenderlab`. MovieLense has user ratings for movies ranging from 1 to 5, where 5 means excellent.

We can load the data by using `data()` function.

```
data("MovieLense")
```

```
class(MovieLense)
```

```
## [1] "realRatingMatrix"
## attr(,"package")
## [1] "recommenderlab"
```

We have never seen an object of class `realRatingMatrix` before. This is because this is a class that is defined by the package `recommenderlab`. Let's take a look at the structure.

```
str(MovieLense, vec.len = 2)
```

```
## Formal class 'realRatingMatrix' [package "recommenderlab"] with 2 slots
##   ..@ data     :Formal class 'dgCMatrix' [package "Matrix"] with 6 slots
##   .. .. ..@ i       : int [1:99392] 0 1 4 5 9 ...
##   .. .. ..@ p       : int [1:1665] 0 452 583 673 882 ...
##   .. .. ..@ Dim     : int [1:2] 943 1664
##   .. .. ..@ Dimnames:List of 2
##   .. .. .. ..$ : chr [1:943] "1" "2" ...
##   .. .. .. ..$ : chr [1:1664] "Toy Story (1995)" "GoldenEye (1995)" ...
##   .. .. ..@ x       : num [1:99392] 5 4 4 4 4 ...
##   .. .. ..@ factors : list()
##   ..@ normalize: NULL
```

`MovieLense` looks like a list but we usually reference the list elements using `\\$` sign rather than `\\@` sign. So what seems to be different about `realRatingmatrix`? According to the documentation, `recommenderlab` is implemented using formal classes in the S4 class system.[2] We can formally check it using `isS4()` function from base R.

```
isS4(MovieLense)
```

```
## [1] TRUE
```

As this is a new class for us, it is important to understand the methods that are applicable to this class. We can use `methods()` function from `utils` package in base R to achieve this.

```
methods(class = class(MovieLense))
```

```
##  [1] [                     [<-                   binarize
##  [4] calcPredictionAccuracy coerce               colCounts
##  [7] colMeans              colSds                colSums
## [10] denormalize           dim                   dimnames
## [13] dimnames<-            dissimilarity         evaluationScheme
## [16] getData.frame         getList               getNormalize
## [19] getRatingMatrix       getRatings            getTopNLists
## [22] image                 normalize             nratings
## [25] Recommender           removeKnownRatings    rowCounts
## [28] rowMeans              rowSds                rowSums
## [31] sample                show                  similarity
## see '?methods' for accessing help and source code
```

Looks like `realRatingMatrix` has several methods associated with it. For example, it can calculate row and column counts directly by using `rowCounts()` and `colCounts()` functions, respectively. Similarly, there is `normalize()` function, which can be used to mean center user ratings.

[2] Read more about S4 class here: http://adv-r.had.co.nz/OO-essentials.html

9.4 Explore `data` from `MovieLense`

```
dim(MovieLense@data)
```

```
## [1]  943 1664
```

MovieLense has ratings by 943 people on 1,664 movies.

```
# Total number of ratings. Has to match the total number of cells (943*1664 =
↪  1,569,152)
sum(table(as.vector(MovieLense@data)))
```

```
## [1] 1569152
```

Let's now check what kind of ratings are contained in `data` matrix.

```
table(as.vector(MovieLense@data))
```

```
##
##       0       1       2       3       4       5
## 1469760    6059   11307   27002   33947   21077
```

The rating corresponding to 0 is actually a missing rating. Indeed, this is a sparse matrix with only about 99,000 actual ratings and rest 1.47 million cells with missing values.

9.4.1 Get some idea about the average ratings of the movies

Figure 9.4 shows the distribution of the average movies ratings. Although the distribution looks mostly bell-shaped, there are spikes at the extremes. This could be because these movies did not have enough ratings.

```
colMeans(MovieLense) %>%
  tibble::enframe(name = "movie",
                  value = "movie_rating") %>%
  ggplot(aes(movie_rating)) +
  geom_histogram(color = "white") +
  theme_minimal()
```

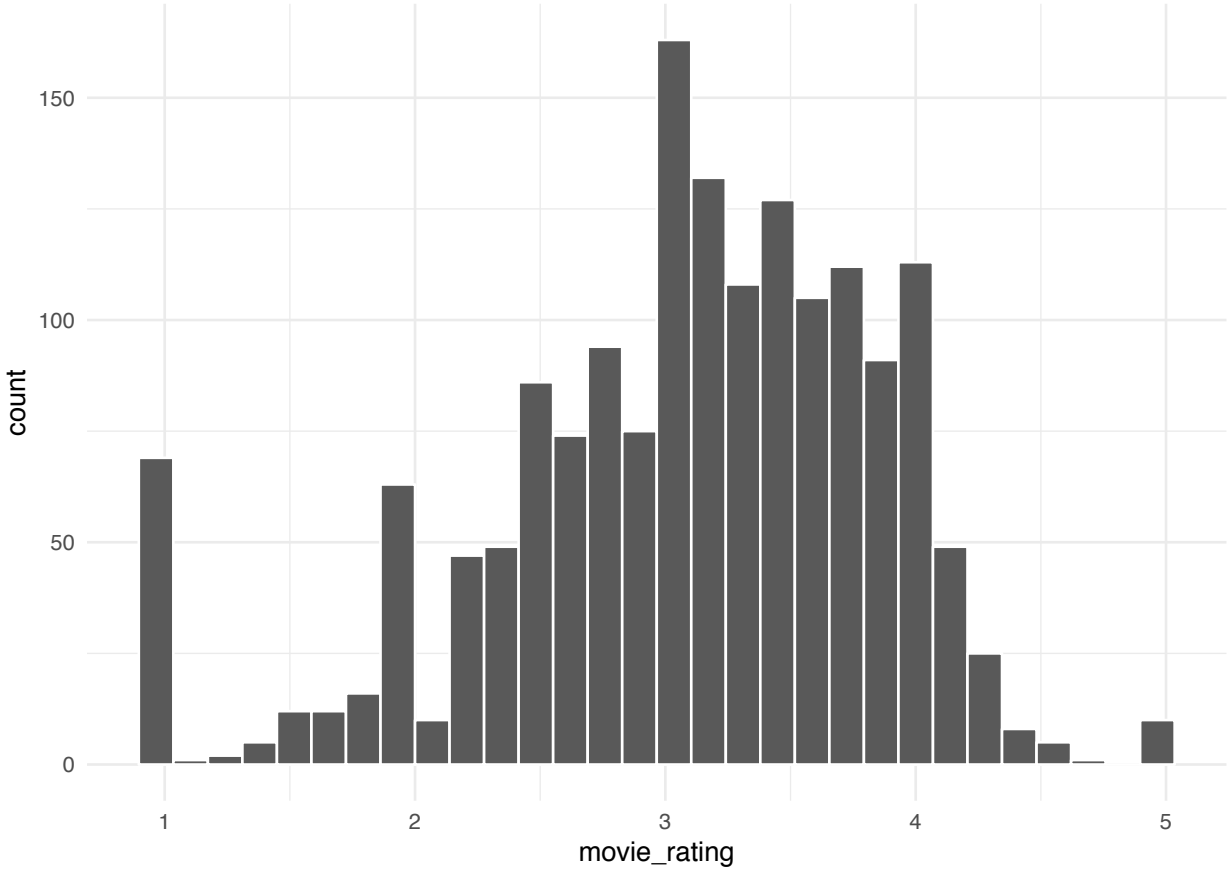


Figure 9.4: Average Movie Ratings

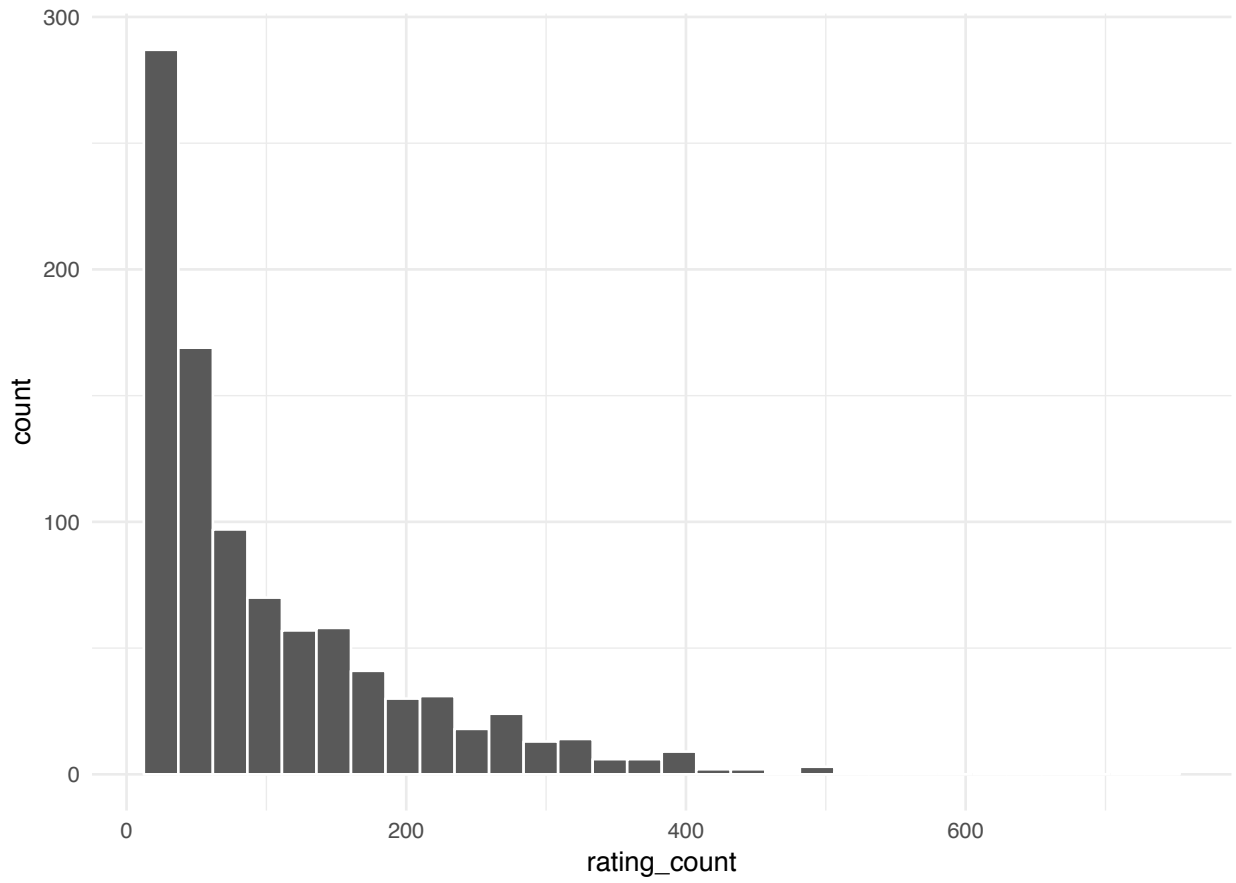


Figure 9.5: User Rating Count

9.4.2 Get some idea about the total number of ratings by each user

Figure 9.5 shows the distribution of the user rating count. As expected, most users do not rate many movies. This is consistent with the power law in user rating counts in other domains.[3]

```
rowCounts(MovieLense) %>%
    tibble::enframe(name = "user",
                    value = "rating_count") %>%
  ggplot(aes(rating_count)) +
  geom_histogram(color = "white") +
  theme_minimal()
```

For the analysis it might be a good idea to remove extreme movies and users. If a movie is rated by less than 50 users then we will drop it. If a user has rated fewer than 25 movies we will drop the user. Obviously these are subjective cutoffs so you can play around with other values. We will create a smaller matrix `movie_small` with these filters.

[3]Read http://www.shirky.com/writings/powerlaw_weblog.html

```
movie_small <- MovieLense[rowCounts(MovieLense) ≥ 25,
                      colCounts(MovieLense) ≥ 50]
movie_small
```

```
## 816 x 601 rating matrix of class 'realRatingMatrix' with 80921 ratings.
```

By using these cutoffs, we shrunk our data considerably. The resulting matrix is only about 31% of the original matrix.

```
sum(table(as.vector(movie_small@data))) /
  sum(table(as.vector(MovieLense@data)))
```

```
## [1] 0.3125357
```

9.5 Build a recommender

We build the recommender with an evaluation scheme using `evaluationScheme()` function. Here we provide all the relevant information for creating a recommender in the next step. Consider this as something similar to `trainControl()` function from `caret`. A critical difference is that in `evaluationScheme()` we also provide the data, which in our case is `movie_small`. This is because `evaluationScheme()` creates data partitions based on the method that we select.

I personally prefer to use cross validation while building a machine learning model. In the code below, you can change the `method` argument to other values as given in the documentation. `k` specifies the number of cross validation folds. The next argument `given` is a critical parameter. Here we specify how many rating could be used (or withheld) from the test set while validating the model. For example, `given = 15` means that while testing the model, use only randomly picked 15 ratings from every user to predict the unknown ratings. A negative value of `given` specifies the ratings to withhold. For instance, `given = -5` will use all the rating except 5 ratings for every user to test the model.

All else equal, a model that performs well with lower values of `given` is desirable because user ratings are sparse.

Finally, pick a threshold for `goodRating`, which will be used for recommending the movies later on. I have picked 4 in the code below, meaning any movie with a rating 4 and above should be considered as a movie with good rating.

```
set.seed(12345)
eval_movies <- evaluationScheme(data = movie_small,
                      method = "cross-validation",
                      k = 10,
                      given = 15,
                      goodRating = 4)
eval_movies
```

```
## Evaluation scheme with 15 items given
## Method: 'cross-validation' with 10 run(s).
## Good ratings: ⩾4.000000
## Data set: 816 x 601 rating matrix of class 'realRatingMatrix' with 80921 ratings.
```

`evaluationScheme()` creates 3 data sets. It splits the data into train and test set but then within the test set it further creates a `known` and an `unknown` data sets. The `known` test data has the ratings specified by `given` and unknown has the remaining ratings, which will be used to validate the predictions made using `known`.

For ease of exposition below, we save these data sets separately.

```
train_movies <- getData(eval_movies, "train")
known_movies <- getData(eval_movies, "known")
unknown_movies <- getData(eval_movies, "unknown")
```

9.6 Evaluate recommender performance

Now we are all set to build and test various recommenders. `recommenderlab` gives us several options. We will test all of them. These are specified using `method` argument and the possible values are: `IBCF`, `UBCF`, `POPULAR`, `RANDOM`, `SVD` and `SVDF`. `RANDOM` just presents random items to users and works as a benchmark for model comparisons.

The model evaluation in this case will be done using `RMSE` and similar metrics. Lower values suggest better model performance.

9.6.1 IBCF

```
ibcf <-
  train_movies %>%
  Recommender(method = "IBCF")

ibcf_eval <- ibcf %>%
  predict(known_movies, type = "ratings") %>%
  calcPredictionAccuracy(unknown_movies)
```

Print the model stats.

```
print(ibcf_eval)
```

```
##      RMSE      MSE      MAE
## 1.526054 2.328841 1.159514
```

9.6.2 UBCF

```
ubcf <-
  train_movies %>%
  Recommender(method = "UBCF")

ubcf_eval <- ubcf %>%
  predict(known_movies, type = "ratings") %>%
  calcPredictionAccuracy(unknown_movies)
```

Print the model stats.

```
print(ubcf_eval)
```

```
##      RMSE       MSE       MAE
## 1.0304528 1.0618330 0.8246746
```

9.6.3 Popular

```
pop <-
  train_movies %>%
  Recommender(method = "POPULAR")

pop_eval <- pop %>%
  predict(known_movies, type = "ratings") %>%
  calcPredictionAccuracy(unknown_movies)
```

Print the model stats.

```
print(pop_eval)
```

```
##      RMSE       MSE       MAE
## 0.9626323 0.9266609 0.7606155
```

9.6.4 Random

```
random <-
  train_movies %>%
  Recommender(method = "RANDOM")
```

```
random_eval <- random %>%
  predict(known_movies, type = "ratings") %>%
  calcPredictionAccuracy(unknown_movies)
```

Print the model stats.

```
print(random_eval)
```

```
##     RMSE      MSE      MAE
## 1.382583 1.911535 1.092926
```

9.6.5 SVD

```
svd <-
  train_movies %>%
  Recommender(method = "SVD")

svd_eval <- svd %>%
  predict(known_movies, type = "ratings") %>%
  calcPredictionAccuracy(unknown_movies)
```

Print the model stats.

```
print(svd_eval)
```

```
##      RMSE       MSE       MAE
## 1.0361175 1.0735394 0.8315897
```

9.6.6 SVDF

This might take some time depending on your processor.

```
svdf <-
  train_movies %>%
  Recommender(method = "SVDF")


svdf_eval <- svdf %>%
  predict(known_movies, type = "ratings") %>%
  calcPredictionAccuracy(unknown_movies)
```

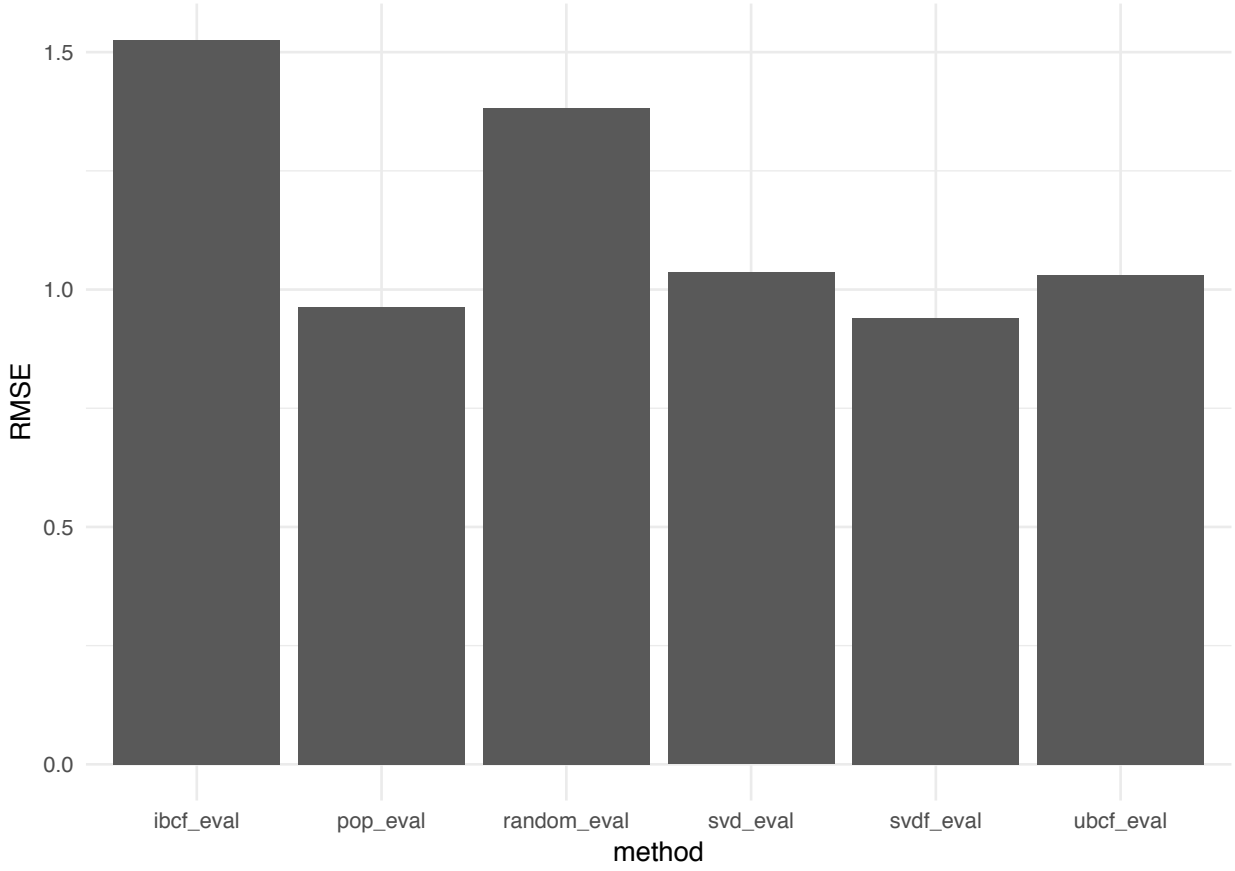


Figure 9.6: Recommender Performance

Print the model stats.

```
print(svdf_eval)
```

```
##       RMSE       MSE       MAE
## 0.9396889 0.8830153 0.7413634
```

We can now plot the `RMSE` of all the recommenders we built.

```
rbind(ibcf_eval, ubcf_eval, pop_eval,
      random_eval, svd_eval, svdf_eval) %>%
  as.data.frame() %>%
  tibble::rownames_to_column(var = "method") %>%
  ggplot(aes(x = method, y = RMSE)) +
  geom_col() +
  theme_minimal()
```

Figure 9.6 shows the performance of each recommender algorithm. In our case, popular method

Figure 9.7: Netflix Popular

has done well. This suggests that recommending people the most popular movies is a good option in this data. This is not surprising as Netflix often shows us the popular movies or TV shows. Figure 9.7 shows such recommendations from Netflix.

9.7 Giving recommendations

Finally, we will use the "popular" method to provide recommendations. We will show top 5 movies to the first 4 users.

```
recos_pop <- pop %>%
  predict(known_movies, n = 5)
```

```
as(recos_pop, "list") %>%
  head(4)
```

```
## $`2`
## [1] "Star Wars (1977)"                "Raiders of the Lost Ark (1981)"
## [3] "Fargo (1996)"                    "Silence of the Lambs, The (1991)"
## [5] "Schindler's List (1993)"
##
## $`27`
## [1] "Raiders of the Lost Ark (1981)"   "Godfather, The (1972)"
## [3] "Silence of the Lambs, The (1991)" "Schindler's List (1993)"
## [5] "Shawshank Redemption, The (1994)"
##
## $`63`
## [1] "Star Wars (1977)"                "Raiders of the Lost Ark (1981)"
## [3] "Godfather, The (1972)"           "Fargo (1996)"
## [5] "Silence of the Lambs, The (1991)"
##
## $`73`
## [1] "Star Wars (1977)"                "Raiders of the Lost Ark (1981)"
## [3] "Godfather, The (1972)"           "Fargo (1996)"
```

```
## [5] "Silence of the Lambs, The (1991)"
```

As expected, popular recommendations are the same for all the users.

Let's also see the recommendations based on Funk SVD.

```
recos_svdf <- svdf %>%
  predict(known_movies, n = 5)
```

```
as(recos_svdf, "list") %>%
  head(4)
```

```
## $`16`
## [1] "Star Wars (1977)"                 "Shawshank Redemption, The (1994)"
## [3] "Empire Strikes Back, The (1980)"  "Godfather, The (1972)"
## [5] "Good Will Hunting (1997)"
##
## $`26`
## [1] "Titanic (1997)"                  "Empire Strikes Back, The (1980)"
## [3] "Star Wars (1977)"                "Raiders of the Lost Ark (1981)"
## [5] "Braveheart (1995)"
##
## $`44`
## [1] "Pretty Woman (1990)"            "Forrest Gump (1994)"
## [3] "American President, The (1995)" "Field of Dreams (1989)"
## [5] "Independence Day (ID4) (1996)"
##
## $`53`
## [1] "Schindler's List (1993)"          "Casablanca (1942)"
## [3] "Shawshank Redemption, The (1994)" "It's a Wonderful Life (1946)"
## [5] "Wrong Trousers, The (1993)"
```

Funk SVD suggests different movies to different users.

9.8 Summary

This chapter introduces the concept of collaborative filtering, which is widely used in recommender systems. Using `MovieLense` data set from `recommenderlab` package, we build a movie recommender. We compare multiple methods and find that recommendations based on popular movies and Funk SVD have the least RMSE.

Chapter 10

Topic Modeling

Topic modeling allows us to identify topics embedded in the textual data. The assumption is that each text document is composed of one or more topics, which are latent. The job of a topic model then is to extract the topics from text. In this chapter we will use a popular probabilistic model called Latent Dirichlet Allocation (LDA) first introduced by Blei, Ng, and Jordan (2003).[1] LDA is a unsupervised machine learning method as we don't know the target variable (i.e., the latent topic) ex ante.

10.1 Latent Dirichlet Allocation

Imagine that you are in your dentist's waiting room. You pick up a magazine and casually start browsing it. While eyeballing text on random pages, you came across the following text:

Williamson and Taylor then began to reap the rewards of their patience and accelerated their way to an immensely productive partnership of 160 from 28.5 overs before Taylor chipped to Jason Holder at mid-on off the bowling of Chris Gayle to depart for 69. Williamson remained steady as ever and went on to record his second consecutive hundred, following on from his knock against South Africa on Wednesday. The Kiwi skipper eventually fell for 148, amassing over half of his side's final total of 291. Cottrell was West Indies' leading man, finishing with figures of 4/56.

With Evin Lewis suffering an injury to his hamstring, Shai Hope joined Chris Gayle at the crease to begin the chase, but the right-hander perished early to Trent Boult, and Nicholas Pooran followed him back to the sheds not long after. Gayle took a liking to Matt Henry's bowling and his partnership with Shimron Hetmyer – featuring some monstrous sixes – saw West Indies take control of the match. The game then swung back in the Black Caps' favour as Lockie Ferguson interrupted with the removal of Hetmyer with an incredible slower ball that initiated a collapse of five wickets for 22 runs.[2]

Unless you are from the UK, Indian Subcontinent, Australia, New Zealand, South Africa, or the Caribbean, there is little chance you understood anything meaningful in this text! However, some of these words look familiar to you: **ball**, **total**, **match**, **game**, and **runs**. These words give you a hint about the topic underlying this text. This looks like a sports article discussing a game

[1] Blei, David M (2003), "Latent Dirichlet Allocation," Journal of Machine Learning Research, 3, 993–1022.

[2] Source: https://www.cricketworldcup.com/news/en/1253879

between South Africa and West Indies. Indeed, after some Google search, you find out that this is game of Cricket.

Note that in order to determine the topic underlying the text, you used some of the words in the text. In an article that describes a game of Cricket, it is likely that the article will use words associated with Cricket. However, some of these words also may appear in an article about Baseball. So there is some uncertainty in your mind about the topic of the text. You decide to assign 60-40 probabilities to Cricket and Baseball.

LDA works on a similar principle. It assumes a data generating process under which a document is generated based on a mix of latent topics and the words that pertain to those topics. As such LDA treats an article as a "bag of words". LDA ignores the ordering of those words. Thus, for LDA both these sentences are the same:

Williamson remained steady as ever and went on to record his second consecutive hundred, following on from his knock against South Africa on Wednesday.

and

steady remained Williamson as ever and record on to went his hundred second consecutive Wednesday against Africa South on, knock following on from his.

LDA assumes that each document has a set of topics, which follow multinomial logistic distribution. However, the probabilities of the multinomial model are not fixed for all the documents. LDA assumes that the distribution of probabilities follow Dirichlet distribution. Thus, for each document, the topics are random draws from a multinomial distribution. The probabilities of the multinomial distribution are in turn are random draws from Dirichlet distribution. The choice of this distribution is due to mathematical convenience as Dirichlet distribution is a conjugate prior to multinomial logistic distribution. As a result, the ***posterior*** distribution of the probability distribution is Dirichlet too. This significantly simplifies the inference problem.[3]

Our task in topic modeling using LDA can be broken down in the following steps:

1. Create a corpus of multiple text documents.
2. Preprocess the text to remove numbers, stop words, punctuation, etc. Additionally, use stemming.
3. Decide the number of topics and fit LDA on the corpus. The number of topics is a hyperparameter to tune.
4. Get the most common words defining each topic. Give them meaningful labels.

10.2 Data

Load/install the packages as follows.

In the classroom if `qdap` fails to install, ignore it for the time being.

[3]For a partial mathematical treatment please refer to the original paper cited above.

```
pacman::p_load(dplyr,
               ggplot2,
               tm, # For textmining
               topicmodels, # For LDA
               qdap, # For some text cleaning
               caret # For random forest
               )
```

We will use a Kaggle dataset consisting of 34,000 Amazon product reviews such as Kindle, Fire TV Stick, etc., provided by Datafiniti:

https://www.kaggle.com/datafiniti/consumer-reviews-of-amazon-products

Download `1429_1.csv` file from Kaggle. You can also download this file from my Github repository: https://github.com/ashgreat/datasets/blob/master/1429_1.csv.zip. As the file size is 49 MB, I recommend downloading the zip file first to your computer and then reading it.

```
reviews <- read.csv("1429_1.csv",
                    stringsAsFactors = FALSE)
```

Take a look at the column names

```
names(reviews)
```

```
##  [1] "id"                 "name"                "asins"
##  [4] "brand"              "categories"          "keys"
##  [7] "manufacturer"       "reviews.date"        "reviews.dateAdded"
## [10] "reviews.dateSeen"   "reviews.didPurchase" "reviews.doRecommend"
## [13] "reviews.id"         "reviews.numHelpful"  "reviews.rating"
## [16] "reviews.sourceURLs" "reviews.text"        "reviews.title"
## [19] "reviews.userCity"   "reviews.userProvince" "reviews.username"
```

We are interested in `reviews.text`. Next we will delete all the rows where the number of words in the reviews were less than 20. You can change this number of something else depending on your application.

Don't run this chunk if you could not install `qdap`

```
reviews <- reviews %>%
  filter(qdap::word_count(.$reviews.text, byrow = TRUE) >= 20)
```

We are left with 18,123 rows.

10.3 Pre-processing

We will extract only the review text.

```
review_text <- reviews %>% pull(reviews.text)
```

10.3.1 Get stop words

I have compiled a large list of stop words from various sources. The list consists of 1,182 stop words. You can download them from my Github repository: https://github.com/ashgreat/datasets/blob/master/my-stopwords.rds?raw=true

```
my_stopwords <- readRDS(gzcon(url("http://bit.ly/2X1Wv2x")))
```

Take a look at some of the stop words

```
head(my_stopwords, 10)
```

```
##  [1] "a"       "about"   "above"   "after"   "again"   "against" "ain"
##  [8] "all"     "am"      "an"
```

10.3.2 tm package

`tm` is a powerful package for text processing. For it to operate, we will first create a corpus of all the documents. Next, we will use `tm_map` function to remove stop words, numbers, punctuation, white space. Finally we will also stem the words.

```
rev_corpus <- tm::Corpus(VectorSource(review_text)) %>%
  tm_map(content_transformer(tolower)) %>%
  tm_map(removeWords, c(my_stopwords, "amazon")) %>%
  tm_map(removeNumbers) %>%
  tm_map(stripWhitespace) %>%
  tm_map(removePunctuation, preserve_intra_word_dashes = TRUE)%>%
  tm_map(stemDocument)
```

10.4 Document term matrix

Create a document term matrix (DTM) such that it shows the frequency of each word for each document. The DTM rows will have 18,123 documents and the columns will have the unique words. The argument `bounds` in the code below specifies dropping the terms that appear in documents fewer than the lower bound and more than the upper bound. This is instead of using TF-IDF, which is an alternative.

Thus, if a word appears in less than 100 documents or more than 1000 documents, we will drop it.

```
dtm <- rev_corpus %>%
  DocumentTermMatrix(control = list(bounds = list(global = c(100, 1000))))
```

With this, we have just 432 words left in the DTM. Perhaps this is too small for the analysis. If you feel so, you could change the bounds.

```
dim(dtm)
```

```
## [1] 18123   432
```

10.4.1 Remove empty documents

For a few documents, all the frequencies in the corresponding rows are 0. We will get rid of these documents. First, we create an index which will hold the information on the rows to keep. This will be a logical vector.

```
index_kp <- rowSums(as.matrix(dtm)) > 0
```

Check how many rows we will keep.

```
sum(index_kp)
```

```
## [1] 17992
```

So, we are dropping 18,123 - 17,992 = 131 documents. Next, we will adjust `dtm` and `review_text` so that they each have 17,992 rows/elements.

```
dtm <- dtm[index_kp, ]
review_text <- review_text[index_kp]
```

10.5 Fitting the LDA

Now we are ready to fit LDA on our text. For this we will use `dtm`. We have to decide on the number of topics. As this is a hyperparameter we have to tune, we can use a grid search. The optimum value of number of topics will give us the maximum likelihood. I don't go into the details of this, but if you are interested, I recommend checking out Martin Ponweiser's thesis where he provides the method and the R code to do this [PDF]: http://epub.wu.ac.at/3558/1/main.pdf.

`LDA()` gives us the method choice between Variational Expectation-Maximization (VEM) algorithm and Gibb's sampler. We will use the latter. We select `alpha` equal to 0.2. Lower values of `alpha` lead to selecting only a few topics per document. Higher values of `alpha` give us diffused

distribution. This is also a hyperparameter you might want to tune using grid search. We select 2,000 iterations out of which the first 1,000 are not used (that's why "burn-in").

The following code took about 30 seconds on my computer.

```
lda_model <- LDA(x = dtm,
                 k = 20,
                 method = "Gibbs",
                 control = list(seed = 5648,
                                alpha = 0.2,
                                iter = 2000,
                                burnin = 1000)
                 )
```

10.6 LDA output

Now we are ready see the words associated with each of the 20 topics. Recall that we used stemming, which means some of the words will be difficult to read. We print first 10 words for each topic using `terms()` function.

```
terms(lda_model, 10)
```

```
##       Topic 1    Topic 2    Topic 3     Topic 4    Topic 5    Topic 6
##  [1,] "christma" "googl"    "speaker"   "internet" "memori"   "stick"
##  [2,] "wife"     "store"    "sound"     "email"    "card"     "box"
##  [3,] "daughter" "android"  "dot"       "web"      "black"    "stream"
##  [4,] "enjoy"    "instal"   "qualiti"   "surf"     "storag"   "cabl"
##  [5,] "happi"    "limit"    "tap"       "brows"    "friday"   "remot"
##  [6,] "mom"      "appl"     "bluetooth" "check"    "expand"   "roku"
##  [7,] "mother"   "expect"   "listen"    "basic"    "download" "faster"
##  [8,] "son"      "download" "portabl"   "want"     "space"    "fast"
##  [9,] "like"     "load"     "voic"      "video"    "movi"     "netflix"
## [10,] "birthday" "function" "connect"   "access"   "add"      "cut"
##       Topic 7   Topic 8   Topic 9     Topic 10  Topic 11  Topic 12
##  [1,] "prime"   "turn"    "version"   "charg"   "batteri" "smart"
##  [2,] "free"    "issu"    "model"     "connect" "life"    "control"
##  [3,] "movi"    "star"    "upgrad"    "wifi"    "long"    "voic"
##  [4,] "account" "reason"  "paperwhit" "problem" "charg"   "turn"
##  [5,] "access"  "bit"     "older"     "charger" "hour"    "hous"
##  [6,] "video"   "review"  "featur"    "issu"    "week"    "room"
##  [7,] "stream"  "figur"   "origin"    "plug"    "last"    "command"
##  [8,] "member"  "problem" "reader"    "unit"    "day"     "skill"
##  [9,] "netflix" "button"  "improv"    "power"   "hold"    "autom"
## [10,] "content" "open"    "generat"   "replac"  "longer"  "integr"
##       Topic 13   Topic 14 Topic 15   Topic 16  Topic 17    Topic 18
```

```
##  [1,] "weather"  "day"    "parent"   "qualiti" "reader"    "size"
##  [2,] "question" "sale"   "control"  "pictur"  "paperwhit" "small"
##  [3,] "list"     "order"  "daughter" "nice"    "white"     "cover"
##  [4,] "answer"   "review" "child"    "camera"  "night"     "carri"
##  [5,] "news"     "decid"  "son"      "excel"   "paper"     "fit"
##  [6,] "fun"      "store"  "children" "clear"   "eye"       "case"
##  [7,] "shop"     "servic" "case"     "pretti"  "bright"    "hold"
##  [8,] "listen"   "want"   "learn"    "high"    "librari"   "nice"
##  [9,] "timer"    "month"  "download" "fast"    "adjust"    "hand"
## [10,] "joke"     "custom" "age"      "amaz"    "dark"      "travel"
##       Topic 19    Topic 20
##  [1,] "friend"    "ipad"
##  [2,] "famili"    "money"
##  [3,] "user"      "worth"
##  [4,] "enjoy"     "cost"
##  [5,] "learn"     "spend"
##  [6,] "item"      "expens"
##  [7,] "fun"       "extra"
##  [8,] "entertain" "pay"
##  [9,] "technolog" "mini"
## [10,] "member"    "compar"
```

LDA did a fairly good job of picking topics. For instance, Topic 1 is all about celebrations and festivities. Topic 2 seems to be about Google App Store, Android, and download speeds of the apps. Topic 3 is about speaker quality, Topic 4 is about Internet, Topic 5 is about memory and storage, and so on.

Topic 20 looks to be about iPad and how it is worth the money. We can expect that whenever this topic showed up, the reviewer probably rated the corresponding Amazon product lower. We will perform this analysis next.

10.7 Topic distribution

We can get some idea about the incidence of each topic in the corpus by looking at the average posterior distribution of the topics. This is arguably a crude way. We can also plot the probability distributions.

For this, we first need to get the posterior distributions of θ. The function `posterior()` returns posteriors of both θ and β. Whereas θ is the posterior distribution of the topics, β is the posterior distributions of the words or terms.

```
lda_post <- posterior(lda_model)
theta <- lda_post$topics
beta <- lda_post$terms
```

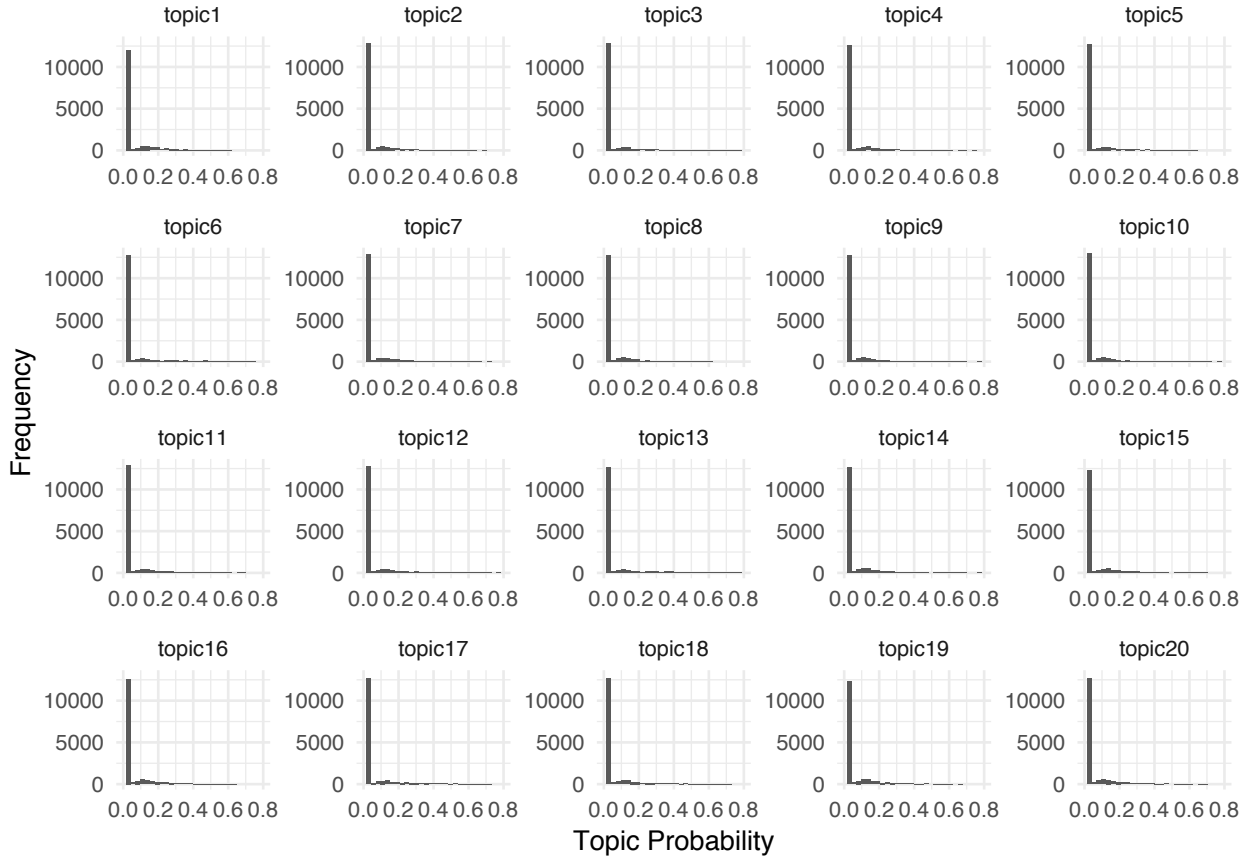
Let's take a look at the average probability of each topic across all the documents.

```
colMeans(theta)
```

```
##          1          2          3          4          5          6
## 0.05307823 0.04700218 0.05099744 0.04934087 0.04734909 0.05803044
##          7          8          9         10         11         12
## 0.04891069 0.04578102 0.04829971 0.04550166 0.04627746 0.05223969
##         13         14         15         16         17         18
## 0.05878948 0.04799089 0.05265680 0.04739848 0.05349202 0.04862603
##         19         20
## 0.04988862 0.04834919
```

We see that the average topic incidence probability is pretty much uniform across the documents. Of course, this hides all the variation. Perhaps it is easier to look at the plots of probabilities. Figure ?? shows the facet plot.

```
theta %>%
  as.data.frame() %>%
  rename_all(~ paste0("topic", 1:20)) %>%
  reshape2::melt() %>%
  ggplot(aes(value)) +
  geom_histogram() +
  scale_x_continuous(limits = c(0, 0.8)) +
  scale_y_continuous(limits = c(0, 13000)) +
  facet_wrap(~ variable, scales = "free") +
  labs(x = "Topic Probability", y = "Frequency") +
  theme_minimal()
```



We don't see a lot of variation in the probability distributions. However, this plot doesn't tell us which topics belong to which documents. If some topics tend to relate closely to certain types of reviews, perhaps we can get an idea about how these topics relate to the reviewer rating. We turn to that next.

10.8 Topic importance

LDA doesn't tell us the importance of each topic because it doesn't know how to determine the importance. However, if some topics relate to glowing reviews while some others are related to bad reviews, perhaps we can determine the topic importance by predicting review rating using the topic probabilities.

I will show you how to do it using Random Forest. There is a strong chance that fitting Random Forest model will take a long time so we will skip doing that in the classroom.

Before we proceed, we need to create a new data set with the review ratings and topic probabilities. Note that some of the `rating` are missing. Furthermore, I convert `rating` into an ordered factor. This is because there are only 5 distinct values of ratings and they are not uniformly distributed. It's easier to treat it as a classification problem.

```
theta_rating <- as.data.frame(theta) %>%
  rename_all(~ paste0("topic", 1:20)) %>%
  mutate(rating = factor(reviews$reviews.rating[index_kp],
                         ordered = TRUE)) %>%
  filter(!is.na(rating))
```

```
class(theta_rating$rating)
```

```
## [1] "ordered" "factor"
```

10.9 Random Forest

In order to fit Random Forest, we first specify the control parameters for training. We will use 10-fold cross-validation with 80-20 train and test split. Furthermore, we perform grid search on 18 values of `mtry`.[4]

```
trControl <- trainControl(method = "cv",
                          number = 10,
                          p = 0.8)

tuneGrid <- expand.grid(mtry = 1:18)
```

Train the model. **It will take several hours so I suggest you use `tuneGrid = expand.grid(mtry = 1)` in the following code.**

```
set.seed(9403)
model_rf <- train(rating ~ . ,
                  data = theta_rating,
                  method = "rf",
                  ntree = 1000,
                  trControl = trControl,
                  tuneGrid = tuneGrid # Use expand.grid(mtry = 1) in the class
                  )
```

Figure 10.1 shows the model accuracy as a function of `mtry`. The highest accuracy is obtained when `mtry = 1`.

```
model_rf$results %>%
  ggplot(aes(mtry, Accuracy)) +
  geom_line() +
  theme_minimal()
```

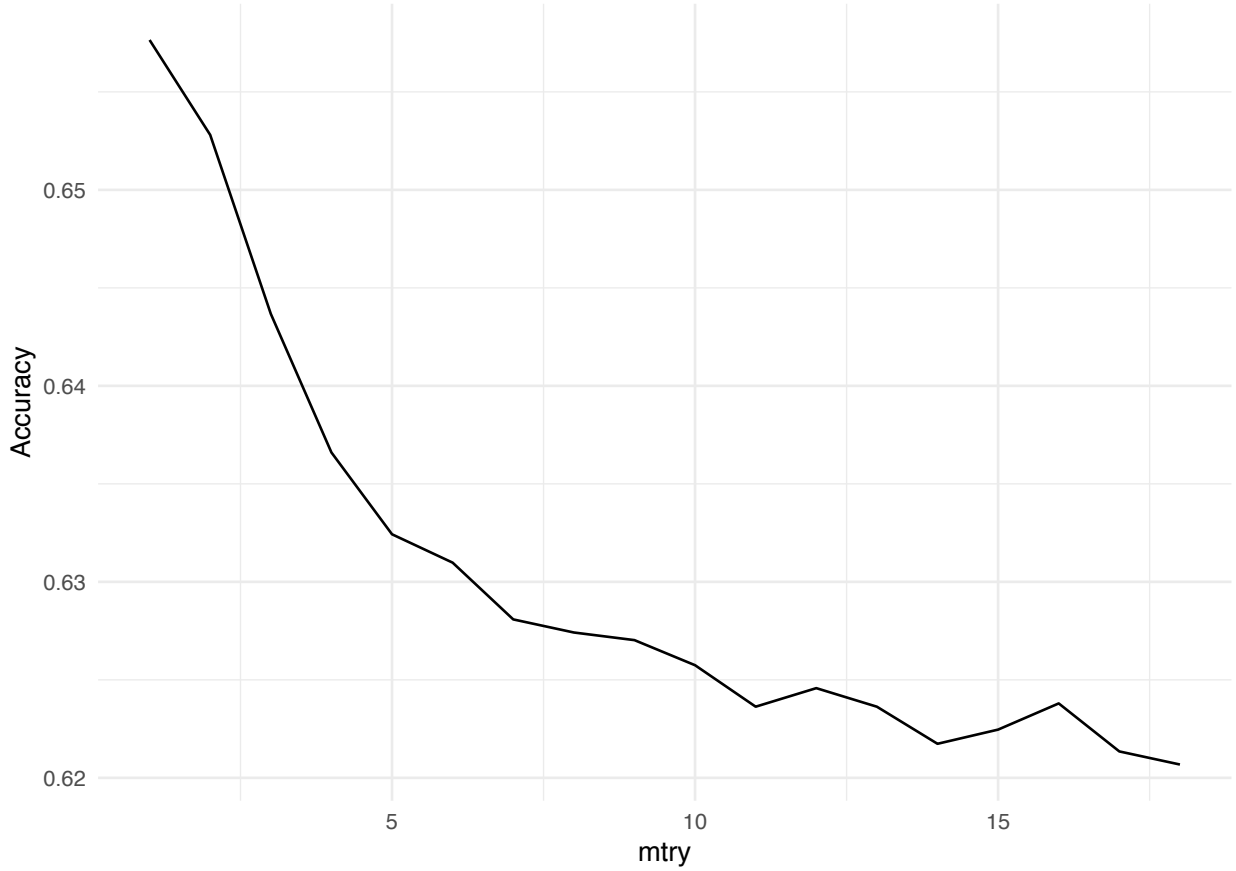


Figure 10.1: Random Forest Tuning

```
model_rf$bestTune
```

```
##   mtry
## 1    1
```

10.9.1 Model performance

Note that we did not split the data into train and test set. As such we will assess model performance on the entire training set. We first get the predicted values of `rating` based on our model.

```
rf_predict <- predict(model_rf, theta_rating)
```

Next, we create a confusion matrix using predicted and observed ratings.

```
confusionMatrix(rf_predict,
                reference = theta_rating$rating)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction     1     2     3     4     5
##          1    91     0     0     0     0
##          2     0    81     0     0     0
##          3     0     0   226     0     0
##          4    11     8    21  1314     6
##          5   176   190   676  3377 11784
##
## Overall Statistics
##
##                Accuracy : 0.7514
##                  95% CI : (0.745, 0.7577)
##     No Information Rate : 0.6564
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.3581
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: 1 Class: 2 Class: 3 Class: 4 Class: 5
## Sensitivity          0.327338  0.29032  0.24485  0.28011   0.9995
```

[4] I estimated several RFs on various LDAs for this example and I ended up with `mtry = 1` as the best hyperparameter in all the cases.

Table 10.1: Random Forest Variable Importance

Topics	Overall
topic2	100.00000
topic8	74.32113
topic10	40.63719
topic6	24.17243
topic17	19.08622
topic14	15.69933
topic4	14.32005
topic5	12.87782
topic20	12.63768
topic7	12.40210

```
## Specificity          1.000000  1.00000  1.00000  0.99653   0.2839
## Pos Pred Value       1.000000  1.00000  1.00000  0.96618   0.7273
## Neg Pred Value       0.989536  0.98893  0.96070  0.79658   0.9966
## Prevalence           0.015478  0.01553  0.05139  0.26118   0.6564
## Detection Rate       0.005067  0.00451  0.01258  0.07316   0.6561
## Detection Prevalence 0.005067  0.00451  0.01258  0.07572   0.9021
## Balanced Accuracy    0.663669  0.64516  0.62243  0.63832   0.6417
```

The classification accuracy of Random Forest model is 75%, which is only marginally better than the no information rate of 66%. The no information rate is the relative frequency of "5" rating. In other words, if we predicted that every review has a 5 rating, we will be correct 66% of times.

10.9.2 Variable importance

Finally, it is time to find out the most important topics. The topics that make the most impact on ratings will be considered important. Table 10.1 shows the variable importance. Topics 2, 8, and 10 are the top 3 important topics.

```
varImp(model_rf, scale = TRUE)
```

In section 10.6 we printed the top 10 words associated with each topic. Let's print them only for Topics 2, 8, and 10 to get a sense of what these topics are.

```
terms(lda_model, 10)[, c(2, 8, 10)]
```

```
##       Topic 2    Topic 8   Topic 10
##  [1,] "googl"    "turn"    "charg"
##  [2,] "store"    "issu"    "connect"
##  [3,] "android"  "star"    "wifi"
##  [4,] "instal"   "reason"  "problem"
##  [5,] "limit"    "bit"     "charger"
```

```
##  [6,] "appl"     "review"  "issu"
##  [7,] "expect"   "figur"   "plug"
##  [8,] "download" "problem" "unit"
##  [9,] "load"     "button"  "power"
## [10,] "function" "open"    "replac"
```

It looks like these 3 topics are about complaints pertaining to the Amazon electronics. Topic 2 is about comparison with Google App Store installations, download etc. Topic 8 seems less about the product and more about the review system. Finally, Topic 10 is about WiFi connectivity and charger issues.

Given the distribution of ratings in the data, where 66% of the reviews have 5 rating, it seems logical that the topics related to complaints are helping the model differentiate between the good and bad ratings. This is because reviews with 5 ratings have various topics but none of them with complaints.

10.10 Summary

In this chapter we learned how to fit Latent Dirichlet Allocation model on textual data. We used Amazon product reviews data from Kaggle and identified 20 topics from the review text. Next we determined the importance of topics by classifying product ratings using the posterior probabilities of the topics. For this we used Random Forest. A linear model is difficult to use for this application because the probabilities of each row add up to 1.

In this example, we used a fixed number of topics. Ideally, we would like to tune this hyperparameters. The chapter references the method to find the optimum number of topics.

Chapter 11

Final Words

We have finished a nice book.