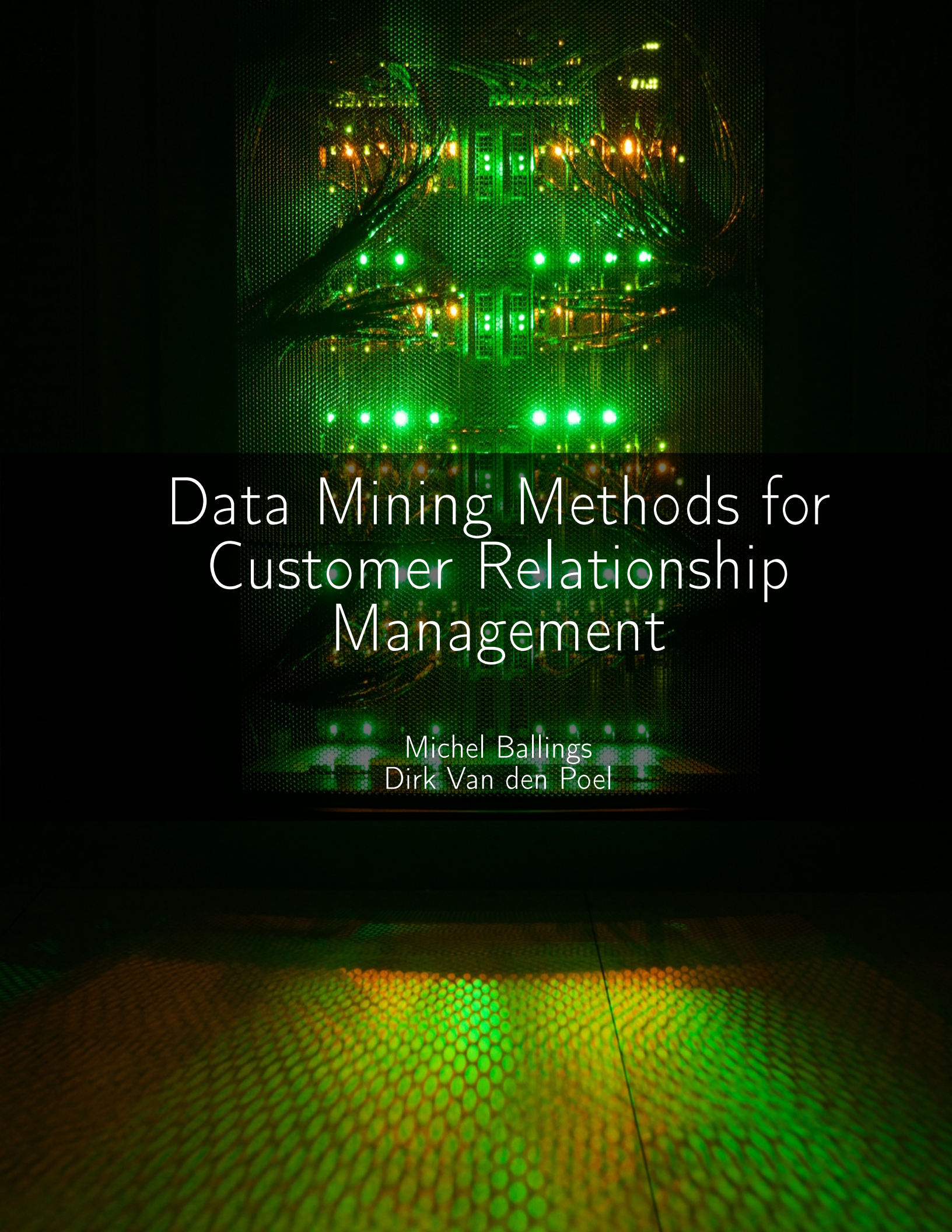


Data Mining Methods for Customer Relationship Management

Michel Ballings
Dirk Van den Poel

Version: 2017/04/11/10:58:12

Preface

The focus of this book is multidisciplinary and lies at the intersection of three disciplines: (1) statistics and machine learning, (2) programming and computing (IT), and (3) business, management and marketing (see Figure 1).

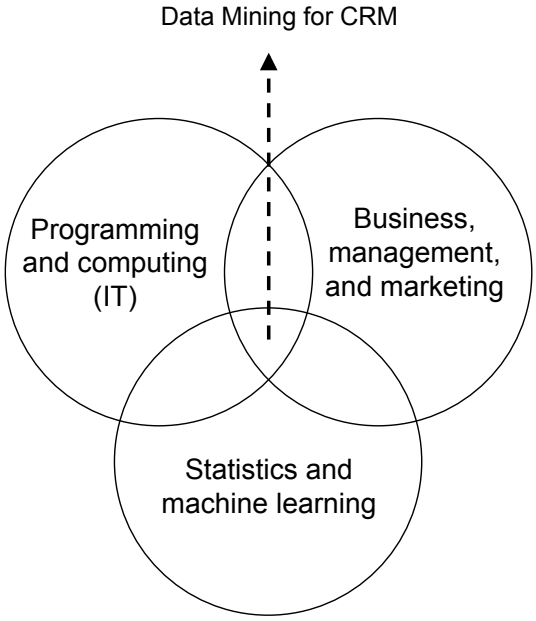


Figure 1: Focus of this book

From the statistics and machine learning side you will learn about various algorithms, how they work, what their parameters are and how to tune them. You will also learn how to evaluate models. From the programming and computing perspective you will learn how to process data in order to make it ready for analysis. Finally you will learn companies' business rationale for using these methods. As you will notice later in this book, it is critical to have a clear understanding of these three components if you want to build a effective data mining applications.

This book can be read in two parts. The first part covers Data Mining methods and software (Chapter 2). While the examples are about Customer Relationship Management (CRM) all the material also applies to other fields such as stock quote prediction, medical screening, credit scoring, and fraud detection. The level of difficulty of exercises will be indicated by stars. One '*' denotes the easiest exercises, and '*****' denotes the most difficult exercises. The second part

(Chapters 3 to 7) brings together all the concepts from the first part in large CRM data mining case studies.

Table of Contents

1 Introduction **9**
1.1 Data Mining in Business . . . 9
1.2 Data Mining in Customer Relationship Management . . . 11

2 Data Mining: Process, Methods, and Software **13**
2.1 Data Mining Process and Software . . . 13
2.2 Business Understanding . . . 15
2.3 Data Understanding . . . 16
2.3.1 Entity relationship diagram . . . 16
2.3.2 Data dictionary . . . 16
2.3.3 Code overview . . . 18
2.3.3.1 Creating, storing and looking at objects . . . 18
2.3.3.2 Object types . . . 18
2.3.3.3 Subsetting . . . 21
2.3.3.4 Memory monitoring and management . . . 26
2.3.3.5 Reading in data . . . 33
2.3.3.6 Data exploration . . . 35
2.3.3.7 Installing and loading packages . . . 46
2.3.3.8 Missing values . . . 51
2.3.3.9 Getting help . . . 53
2.3.4 Problems . . . 53
2.4 Data Preparation . . . 53
2.4.1 Time window . . . 53
2.4.2 Code flowchart . . . 54
2.4.3 Code Overview . . . 55
2.4.3.1 Operators . . . 55
2.4.3.2 Creating dummy variables . . . 59
2.4.3.3 Conditional processing . . . 61
2.4.3.4 Timing code . . . 62
2.4.3.5 Loops . . . 62
2.4.3.6 Applying functions to lists . . . 67
2.4.3.7 Aggregating . . . 69
2.4.3.8 Merging . . . 70

2.4.3.9 Working with dates . . . 73
2.4.3.10 Creating functions . . . 79
2.4.3.11 Text Mining . . . 81
2.4.3.12 Speeding up code with the data.table package . . . 96
2.4.4 Guidelines for basetable creation . . . 108
2.4.4.1 Which customers to include? . . . 108
2.4.4.2 How to compute the response variable? . . . 109
2.5 Modeling . . . 110
2.5.1 Training, validation, and test set . . . 110
2.5.2 Building blocks of algorithms . . . 113
2.5.3 Naive bayes . . . 113
2.5.4 Logistic regression . . . 120
2.5.5 Neural networks . . . 134
2.5.6 K-nearest neighbors . . . 142
2.5.7 Decision trees . . . 149
2.5.8 Support vector machines . . . 159
2.5.9 Ensemble Methods . . . 169
2.5.10 Bagged trees . . . 171
2.5.11 Random forest . . . 173
2.5.12 Kernel factory . . . 178
2.5.13 Adaptive boosting . . . 178
2.5.14 Rotation forest . . . 189
2.6 Model Evaluation . . . 191
2.6.1 Performance measures . . . 191
2.6.1.1 Binary classification models . . . 191
2.6.1.2 Multiclass classification models . . . 203
2.6.1.3 Regression models . . . 203
2.6.2 Cross- Validation . . . 204
2.6.3 Understanding classifier performance . . . 208
2.6.4 Partial dependence plots . . . 216
2.7 Model Deployment . . . 222
2.7.1 Code structure . . . 222
2.7.2 Method dispatching . . . 224
2.7.3 Making functions invisible . . . 227
2.7.4 The '...' argument . . . 227
2.7.5 Environments . . . 230

3 Case study: Customer Acquisition 235
3.1 Business Understanding . . . 235
3.2 Data Understanding . . . 236
3.3 Data Preparation . . . 244
3.3.0.1 Code flowchart . . . 244
3.3.0.2 Code . . . 244

3.4 Modeling . . . 257
3.5 Model Evaluation . . . 258
3.6 Model Deployment . . . 266
3.6.1 Function to create the model . . . 266
3.6.2 Function to create predictions . . . 273

4 Case study: Customer Up-sell 275

5 Case study: Customer Cross-sell 277
5.1 Business Understanding . . . 277
5.2 Data Understanding . . . 278
5.3 Data Preparation . . . 278
5.4 Modeling . . . 278
5.5 Model Evaluation . . . 278
5.6 Model Deployment . . . 278

6 Case study: Customer Defection 279
6.1 Business Understanding . . . 279
6.2 Data Understanding . . . 280
6.3 Data Preparation . . . 284
6.4 Modeling . . . 284
6.5 Model Evaluation . . . 284
6.6 Model Deployment . . . 284

7 Case study: Customer Lifetime Value 285
7.1 Business Understanding . . . 285
7.2 Data Understanding . . . 286
7.3 Data Preparation . . . 286
7.4 Modeling . . . 286
7.5 Model Evaluation . . . 286
7.6 Model Deployment . . . 286

1 Introduction

1.1 Data Mining in Business

The concepts that underlie the field of predictive analytics, data mining and machine learning were developed a long time ago. Table 1.1 contains the evolution of data mining which is characterized by growth in computing power, availability of data, development of machine learning algorithms, and development of statistical software.

Table 1.1: Evolution of predictive analytics

Year	Evolutionary step
Beginning 19th century	Legendre and Gauss publish their method of least squares (earliest form of linear regression)
1936	Fisher proposed linear discriminant analysis (for classification)
1939	Development of cluster analysis
1940s	Logistic regression (various authors)
1941	First digital computer released
1947	Linear programming developed
1957	FORTRAN programming language developed
1963	Development of microchips
1964	Announcement of first IBM mainframe System/360. The initial system had 64kb of RAM memory.
1968	SPSS is released for mainframes.
1970	Codd conceptualizes first relational database

Early 1970s	Generalized linear models (with logistic and linear regression as special cases)
1972	Personal computers are popularized in business. SAS is launched.
1975	The *S* statistical language and Matlab are launched.
End of 1970s	Many more learning techniques are available. However, almost exclusively linear methods, because fitting non-linear relationships was computationally infeasible then (Gareth et al., 2013).
Mid 1980s	Computing technologies had finally improved sufficiently to be able to fit non-linear relationships. Breiman, Friedman, Olshen and Stone introduce classification and regression trees.
1986	Hastie and Tibshirani coined the term generalized additive models for a class of non-linear extensions to generalized linear models.
1989	Launch of the World Wide Web resulting in an explosion of data. The term data mining is introduced.
1996	Freund and Shapiro introduce AdaBoost. Google began as a research project.
1997	*R* is launched (version 0.16) with base functionality (no packages).
2001	Breiman publishes Random Forest, the most powerful algorithm up to date.
2002	Friedman introduces Stochastic Gradient Boosting.
2004	Google introduces MapReduce, a programming paradigm for Big Data. Facebook is launched.
2006	Twitter is launched.
2007	iPhone 1 is launched. NoSQL databases were developed.
2010	First Kaggle competition.
2015	*R* 3.2.3 is released and has 7754 packages.

These developments in technology, data availability, and algorithms provide new opportunities for many business applications. Examples are provided in Table 1.2. While there are four main tasks in data mining: (1) prediction, (2) association rule learning, (3) clustering, and (4) anomaly detection, this book focuses on prediction as that is the main task in Customer Relationship Management (CRM) applications. The other three tasks will be covered in a less formal format and will be given a supporting role. The prediction techniques that are presented in this book are also applicable to the other domains presented in Table 1.2.

Table 1.2: Examples of data mining applications in business

Application	Example
Investing	Which stocks will go up?
Healthcare	Who will need special care in the near future?
Fraud detection	Which transactions are fradulent?
Customer Relationship Management	Which prospects will be most likely to become customers?
Human resources	Which employees will resign?

1.2 Data Mining in Customer Relationship Management

Customer lifecycle

This book covers five applications: acquisition, up-sell, cross-sell, churn, and customer lifetime value modeling. In the following paragraphs we will discuss what those mean and how these modeling applications can lead to business value.

Acquisition modeling. Every company is challenged with getting new customers. In B2B settings a strategy is to buy a list with names and contact information of prospects from a list vendor. A call center then calls these prospects and hopes that they will buy. But these lists are often very long (e.g., one million prospects) and not all prospects are equally likely to buy. The question arises as to which prospects the company should target. To be able to determine that the best strategy is to build an acquisition model. An acquisition model predicts which prospects are most likely to become a customer. This strategy is contingent upon having information about the prospects in the list that is bought and having identical information (i.e., the same variables) about current customers. The goal is then to learn how similar prospects in the bought list are to current customers. Prospects that are most similar are most likely to become a customer and should therefore be targeted. In other words, (1) we will first compute a score [0,1] for each prospect, (2) rank the prospects based on that score from high to low, and (3) target the top n% of those prospects where n is determined by the company's budget (e.g., how many calls can they place?).

Up-sell modeling. Suppose that a company has a sufficient number of customers but wants to increase its profit margins. One strategy is to try to have customers upgrade their service. This strategy is often used by internet service providers. Instead of calling all of their customers, it is generally more efficient to only call those customers that are most likely to upgrade. The goal is then to look at the behavior of customers that upgraded in the past right up until they upgraded, and compare that with the behavior of customers that have not yet upgraded. Customers that have the most similar behavior are then assumed to be most likely to upgrade. In a similar fashion as in the acquisition application we first compute a score [0,1] for each customer that has not yet upgraded, (2) rank those customers based on that score from high to low, and (3) target the top

n% of those customers where n is determined by the company's budget (e.g., how many calls can they place?).

Cross-sell modeling. Another strategy is to increase margins by trying to sell additional products to a given customer. Instead of calling up a customer and proposing a random product we first build a model that predicts which product the customer is most likely to buy. We approach the problem by looking at what customers have bought at time t and what they have bought at time $t+1$. We can then compare the profile of new customers at time t with the profile of past customers at time t and predict which product those new customers are most likely to buy at $t+1$.

Churn modeling. Another problem might present itself when there are too many customers that leave the company. In order to retain them the company might give a promotion across all customers. However, in doing that the company is also subsidizing customers that were going to stay either way and loose valuable income. A way to approach this problem is build a churn model that predicts which customers are most likely to churn in the future. We compare the behavior of customers with customers that have churned in the past and those customers with a profile very similar to churned customers are assumed to also attrite. We first compute a score [0,1] for each customer that represents their likelihood to churn, second we rank the customers based on that score from high to low, and third we target the top n% customers where n is determined by the company's budget.

Customer Lifetime Value (CLV) modeling. A company may also be interested in knowing how valuable each customer will be in the future. The most valuable customers in the future should be pampered. A prediction of future value may also be combined in a churn strategy where only the customers with the highest churn probability and the highest value should be targeted for retention actions. We approach this problem by learning the relationship between customer behavior in the far past with customer behavior (e.g., purchases) in the near past. We can then use that relationship to predict future behavior.

We have prepared for you five case studies with step by step instructions to develop software applications that can be deployed at a company. We will cover in detail business understanding, data understanding, data preparation, modeling, model evaluation and deployment. All case studies are grounded in reality.

2

Data Mining: Process, Methods, and Software

2.1 Data Mining Process and Software

Since the primary goal of aCRM projects is to have models with high predictive performance we will be looking at data mining methodologies. More specifically, we will adhere to the Cross Industry Standard Process for Data Mining (CRISP-DM) which comprises six phases: business understanding, data understanding, data preparation, data modeling, model evaluation, and model deployment (Chapman et al., 2000). Figure 2.1 displays the whole process. Business understanding refers to understanding why we are creating a model from a business perspective. The outcome of this phase is an answer to the question: 'How will this model be used?'. Data understanding refers to generating insight in the structure of the data by looking at relationships between tables, creating a data dictionary and running descriptives. Data preparation refers to the creation of a data set that is ready to apply algorithms to. In this book the level of analysis is the customer or prospect and hence we want one row per customer in the final dataset, called the basetable. Modeling and evaluation entails fitting a model and evaluating its predictive performance. Finally, deployment refers to installing the software at the company and making sure it is operational for what is was designed to do as understood in the business understanding phase. This includes integrating the solution into the existing business/IT environment.

We clearly see that the process is iterative in nature. At some instances we have to go back to the previous stage. The best strategy is to start out small and run through the cycle as quickly as possible by building a very simple model. It is generally more efficient to gradually make the model more complex and evaluate the model at each increase of complexity as opposed to spending a lot of time in computing a lot of predictors that are maybe not necessary in the first place. In our experience a total of 20 cycles is normal.

Because we want to recreate our models multiple times in the future, without any manual work, we need to write code and make custom built programs, as opposed to working in a point

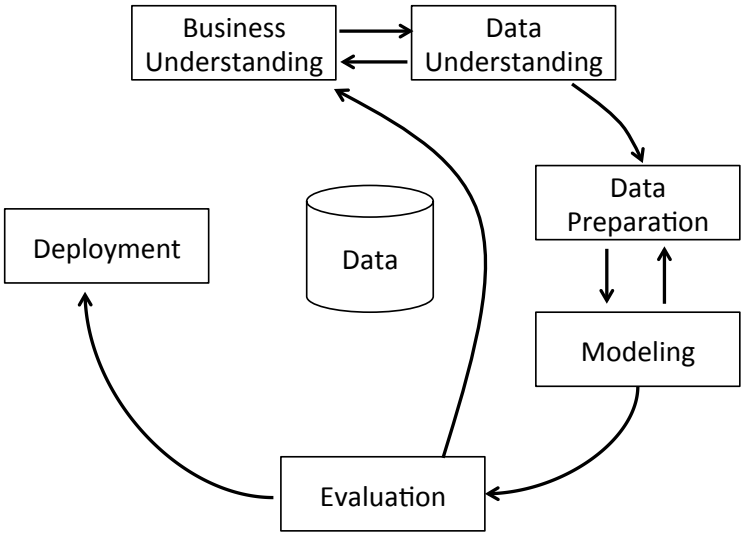


Figure 2.1: CRISP-DM (adapted from Chapman et al. 2000)

and click environment. While there are several software packages that are very good for doing this (e.g., SAS, Oracle DM) we will use R (R Core Team, 2015a). Here are some of R's advantages (Muenchen, 2011):

- **Functionality**. At the time of this writing, R has 7036 available packages (R Core Team, 2015a), including graphics, linear and non-linear modeling, time series analysis, statistical tests, classification, and regression. These packages can be freely downloaded with one line of code in the programming environment itself.
- **Flexibility**. R is not limited to tables and data sets, R has a wide variety of data structures such as vectors, data frames, matrices, and factors (R Core Team, 2015a). This makes it much easier to include variables from different data structures without merging them.
- **Transparency**. R is open- source: one can easily see and change all R- functions (R Core Team, 2015a). Hence one is not limited to documentation and can go and look at the code if something is not clear.
- **Availability**. R is available on almost all operating systems including Microsoft Windows, MacOS X, Linux, and Unix (R Core Team, 2015a).
- **Ease of install**. Precompiled binaries are available for the aforementioned operating systems (R Core Team, 2015a).
- **Cost.** R is free (R Core Team, 2015a).

R can be downloaded from *cran.rstudio.com*. This will give you a bare bones interface. To increase our productivity we like to work in an integrated development environment (IDE) called RStudio Desktop (Open Source Edition), which can be downloaded from *rstudio.com/products/RStudio/#Desktop*.

Figure 2.2 is a screen shot of the environment. We like to have the code development pane (called Source) in the top left, the output of our code in the top right (called Console), the help and plotting areas in the bottom right, and the environment area in the bottom left. This can be modified in Tools > Global Options > Pane Layout. In the beginning new R users will benefit mostly from syntax highlighting which is not available in plain R. Once the user becomes more advanced there are lots of other productivity increasing features. We refer to *rstudio.com* for more information.

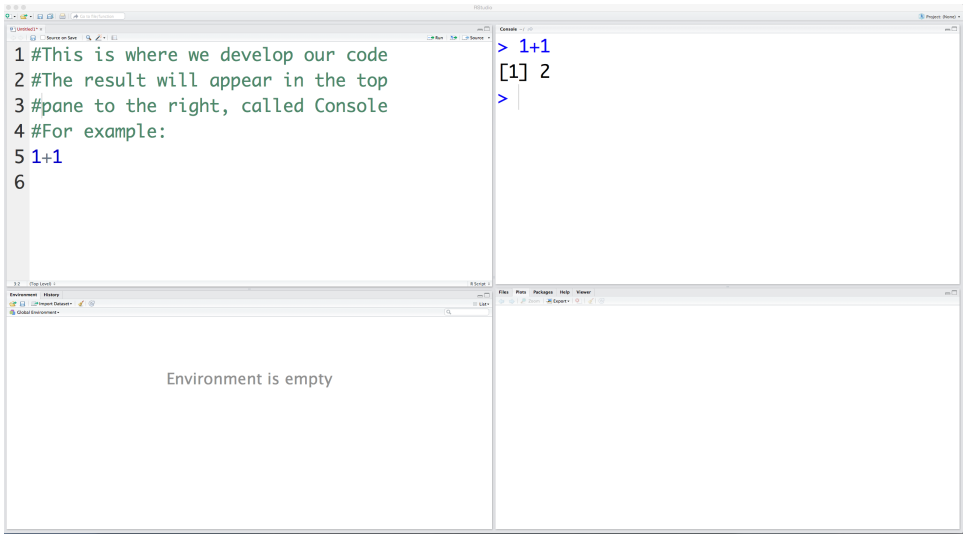


Figure 2.2: RStudio

In terms of R code the objective of all the applications in this book is to develop two functions:

- A function that builds, updates and evaluates models
- A function that deploys the model (i.e., predicts the future)

In the following sections we discuss what the aforementioned six stages entail in more detail. We also provide code examples that one can use in the applications.

2.2 Business Understanding

The business understanding phase consists in understanding the overarching goal from a business perspective (Chapman et al., 2000). Once such an understanding is established, it can be translated into a definition of the data mining problem. Subsequently it is paramount to conceive a preliminary plan that will lead to successful completion of the project's objectives (Chapman et al., 2000).

In this book we will focus on predictive analytics. If we can predict what is going to happen, appropriate action can be taken in advance. The type of applications that we are interested in are screening applications. A screening application looks at a large number of cases, and

identifies which ones are most likely to display a focal behavior. For example, in stock screening, or stock price prediction, the goal is to predict which stocks are going to go up the next day. If we can accurately predict today, which stocks are going to go up tomorrow, we can buy those stocks today and sell them tomorrow with profit. An example in healthcare would be to predict if somebody will need treatment. If we are able to accurately predict which patients are going to have a desease we can call them and take preventive measures. A last example is in customer relationship management. If we can predict which customers are going to cancel their service, we can call them and offer them a promotion before they leave. We need a predictive model for that, since the alternative would be to just let them leave, or offer a promotion to all customers.

2.3 Data Understanding

The data understanding stage consists in exploring the data at the table and variable level. At the table level we need an entity relationship diagram (ERD) that explains how the different tables are related. At the variable level we need a data dictionary. Only proper understanding of the data can result in a high quality basetable and therefore this stage should not be taken lightly. It will speed up any subsequent stages considerably. To generate this proper understanding we may need to create variables, change object types, subset the data, read in the data, run descriptives, and impute missing values. We will also need to look at manual pages, install and load packages and monitor and manage memory usage.

2.3.1 Entity relationship diagram

Figure 2.3 provides an example of an entity relationship diagram (ERD) for three table. Again, there are many different designs of diagrams. The choice of a specific design is driven by preference. We prefer a basic design as displayed in Figure 2.3. Every ERD should display the cardinality (one-to-many, many-to-many, or one-to-one), the variable that is used as the key to merge two datasets and the names of the tables. For example, if we focus in on the relationship between the Subscriptions and Customers table in Figure 2.3 we see that every customer can have one-to-many subscriptions and that every subscription can have exactly one customer. The two tables can be merged by the variable CustomerID. As to the Subscription-Product relationship we see that each subscription can have only one product and that every product can have one-to-many subscriptions. The key to merge on is ProductID.

2.3.2 Data dictionary

There are different degrees of detail that we can use to compile a data dictionary. We choose for a basic level of detail: we use a table of two columns, variable name and definition. This strikes the right balance between amount of work and benefits. Table 2.1 provides an example of a transactions table.

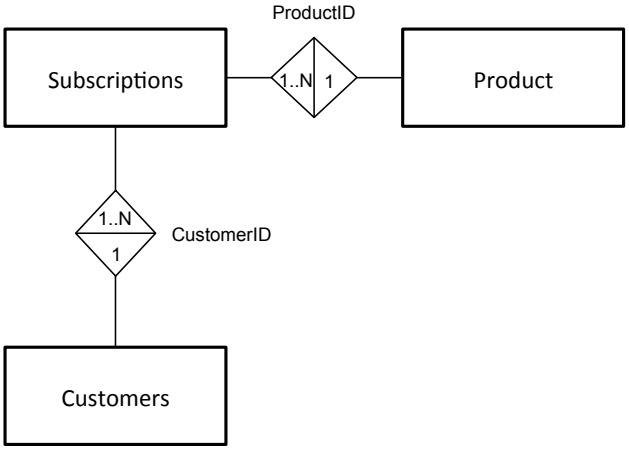


Figure 2.3: Example of an ERD for three tables

Table 2.1: Example of a data description for a transactions table

Variable	Definition
transID	The transaction ID
custID	The customer ID
price	Price paid for the product
prodID	The product ID
purchDt	The date of the purchase

2.3.3 Code overview

All code in this section is available from:
ballings.co/hidden/aCRM/code/chapter2/DataUnd.R

2.3.3.1 Creating, storing and looking at objects

```
#create object and store the value 1
a <- 1

#Whenever you see a function name you don't know,
#just run ?functionname to get more information about it. This also works for
#operators such as "<-". Operators need
#to be quoted when using ?, whereas functions
#do not.For example:
?"<-"

#look at its contents
a

#-# [1] 1

#create, store and look simultaneously
(a <- 2)

#-# [1] 2

#If you want to prevent R from running a line of code, put a # in front of it
#This is called commenting
#(a <- 3)
a

#-# [1] 2

#a is still 2, meaning that #(a <- 3) did not run

# To remove a from the environment use the rm function
rm(a)
#Check to see if a still exists:
a

#-# Error in eval(expr, envir, enclos): object 'a' not found

#It does not exist anymore.
```

2.3.3.2 Object types

There are three main types of objects. The first type is a vector. It is the simplest R data type and is atomic: linear vectors of a single type. The second type is a collection of vectors (matrices

and data frames). Matrices can only contains vectors of the same type whereas data frames can contain vectors of different types. We will most often use data frames. Finally, the third type is a collection of everything (vectors, matrices and data frames).

```
# 1)vectors

# --numeric: decimal numbers
# Let's create a numeric sequence of 26 numbers
(num_vector <- as.numeric(seq(from=1,to=3.5,by=0.1))) #?seq

#-#  [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3
#-# [15] 2.4 2.5 2.6 2.7 2.8 2.9 3.0 3.1 3.2 3.3 3.4 3.5

# --character: strings
(char_vector <- as.character(c("a","b","c","d","e","f","g","h","i","j")))

#-#  [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j"

    # OR    (char_vector <- letters[seq( from = 1, to = 10 )])

# --integer: integer numbers
(int_vector <- as.integer(1:10)) #help(":")

#-#  [1]  1  2  3  4  5  6  7  8  9 10

# --factor: integer but with labels
(fact_vector <- as.factor(c("a","b","c","d","e","f","g","h","i","j")))

#-#  [1] a b c d e f g h i j
#-# Levels: a b c d e f g h i j

#help(c)
str(fact_vector)

#-#  Factor w/ 10 levels "a","b","c","d",..: 1 2 3 4 5 6 7 8 9 10

str(char_vector)

#-#  chr [1:10] "a" "b" "c" "d" "e" "f" "g" ...

# --logical
(log_vector <- c(TRUE,FALSE,TRUE,FALSE))

#-# [1]  TRUE FALSE  TRUE FALSE

str(log_vector)

#-#  logi [1:4] TRUE FALSE TRUE FALSE
```

```r
# 2)collection of vectors:

# --matrix: vectors of same type and length
help(matrix)
(mat <- matrix(data=1:25,nrow=5,ncol=5))

#-#      [,1] [,2] [,3] [,4] [,5]
#-# [1,]    1    6   11   16   21
#-# [2,]    2    7   12   17   22
#-# [3,]    3    8   13   18   23
#-# [4,]    4    9   14   19   24
#-# [5,]    5   10   15   20   25

# --data.frame (what we work with most often):
# vectors can be of different type but same length
(df <- data.frame(int_vector,fact_vector))

#-#    int_vector fact_vector
#-# 1           1           a
#-# 2           2           b
#-# 3           3           c
#-# 4           4           d
#-# 5           5           e
#-# 6           6           f
#-# 7           7           g
#-# 8           8           h
#-# 9           9           i
#-# 10         10           j

# 3)collection of everything
# can be of different type and length

# --list
(l <- list(df,mat,int_vector))

#-# [[1]]
#-#    int_vector fact_vector
#-# 1           1           a
#-# 2           2           b
#-# 3           3           c
#-# 4           4           d
#-# 5           5           e
#-# 6           6           f
#-# 7           7           g
#-# 8           8           h
#-# 9           9           i
#-# 10         10           j
#-#
#-# [[2]]
#-#      [,1] [,2] [,3] [,4] [,5]
#-# [1,]    1    6   11   16   21
```

```
#-# [2,]    2    7   12   17   22
#-# [3,]    3    8   13   18   23
#-# [4,]    4    9   14   19   24
#-# [5,]    5   10   15   20   25
#-#
#-# [[3]]
#-#  [1]  1  2  3  4  5  6  7  8  9 10
```

There is a difference between `as.numeric()` and `numeric()`, `as.character()` and `character()`, and `integer()` and `as.integer()`. The short versions only have one argument (i.e., `length`) and are used to initialize a vector. The long versions have the argument `x` which is the object to be coerced.

In the case of factors this is a little bit different. Both `as.factor()` and `factor()` have the argument `x`, but the latter one has more arguments and is rather used to initialize the vector.

2.3.3.3 Subsetting

Subsetting means the selection of part of the information stored in an object. In the code chunk below we will provide examples for subsetting of vectors, data frames and lists.

```
# 1) subsetting for vectors: Use [indicator]
num_vector

#-#  [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3
#-# [15] 2.4 2.5 2.6 2.7 2.8 2.9 3.0 3.1 3.2 3.3 3.4 3.5

#select the second element
num_vector[2]

#-# [1] 1.1

#select the fist two elements
num_vector[1:2]

#-# [1] 1.0 1.1

#select element 1 and 3
num_vector[c(1,3)]

#-# [1] 1.0 1.2

#Note that '[' is a function,
#whose first argument is the object
#being subsetted
'['(num_vector,c(1,3))

#-# [1] 1.0 1.2
```

```
# 2) subsetting for collection of vectors:
# Use [row indicators, column indicators]
df

#-#    int_vector fact_vector
#-# 1           1           a
#-# 2           2           b
#-# 3           3           c
#-# 4           4           d
#-# 5           5           e
#-# 6           6           f
#-# 7           7           g
#-# 8           8           h
#-# 9           9           i
#-# 10         10           j

#select element on row 3, column 2
df[3,2]

#-# [1] c
#-# Levels: a b c d e f g h i j

#select column 2
df[,2]

#-#  [1] a b c d e f g h i j
#-# Levels: a b c d e f g h i j

df[,c(FALSE,TRUE)]

#-#  [1] a b c d e f g h i j
#-# Levels: a b c d e f g h i j

#select row 2
df[2,]

#-#   int_vector fact_vector
#-# 2          2           b

#remove row 2
df[-2,]

#-#    int_vector fact_vector
#-# 1           1           a
#-# 3           3           c
#-# 4           4           d
#-# 5           5           e
#-# 6           6           f
#-# 7           7           g
#-# 8           8           h
#-# 9           9           i
#-# 10         10           j
```

```
#you can also use the column names
df

#-#    int_vector fact_vector
#-# 1           1           a
#-# 2           2           b
#-# 3           3           c
#-# 4           4           d
#-# 5           5           e
#-# 6           6           f
#-# 7           7           g
#-# 8           8           h
#-# 9           9           i
#-# 10         10           j

#select column 2
df$fact_vector #is the same as df[,2]

#-#  [1] a b c d e f g h i j
#-# Levels: a b c d e f g h i j

df[,"fact_vector"]

#-#  [1] a b c d e f g h i j
#-# Levels: a b c d e f g h i j

#drop=FALSE will ensure that the properties of the object
# are preserved. Always use this inside functions.
df[,2,drop=FALSE]

#-#    fact_vector
#-# 1           a
#-# 2           b
#-# 3           c
#-# 4           d
#-# 5           e
#-# 6           f
#-# 7           g
#-# 8           h
#-# 9           i
#-# 10          j

df[,2]

#-#  [1] a b c d e f g h i j
#-# Levels: a b c d e f g h i j

#to get similar behavior as drop=FALSE with less typing you can do
df[2]
```

```
#-#    fact_vector
#-# 1            a
#-# 2            b
#-# 3            c
#-# 4            d
#-# 5            e
#-# 6            f
#-# 7            g
#-# 8            h
#-# 9            i
#-# 10           j

# However, this is considered bad practice because it makes
# your code less readable and maintainable. If you do not
# have a comma in your brackets the reader assumes the object
# is a vector. Therefore convention is to never use this
# shortcut.

#We can also use a matrix to make our selection
#If want the first three elements of column 1 and
#the last three elements of column 2:
(mat <- matrix(c(c(rep(TRUE,3),rep(FALSE,7)),
               c(rep(FALSE,7),rep(TRUE,3))),ncol=2))

#-#        [,1]  [,2]
#-#  [1,]  TRUE FALSE
#-#  [2,]  TRUE FALSE
#-#  [3,]  TRUE FALSE
#-#  [4,] FALSE FALSE
#-#  [5,] FALSE FALSE
#-#  [6,] FALSE FALSE
#-#  [7,] FALSE FALSE
#-#  [8,] FALSE  TRUE
#-#  [9,] FALSE  TRUE
#-# [10,] FALSE  TRUE

#We can then simply use that matrix within the square brackets:
df[mat]

#-# [1] " 1" " 2" " 3" "h"  "i"  "j"

# 3) subsetting for lists: Use [[]]
l

#-# [[1]]
#-#    int_vector fact_vector
#-# 1           1           a
#-# 2           2           b
#-# 3           3           c
#-# 4           4           d
#-# 5           5           e
```

```
#-# 6           6           f
#-# 7           7           g
#-# 8           8           h
#-# 9           9           i
#-# 10         10           j
#-#
#-# [[2]]
#-#      [,1] [,2] [,3] [,4] [,5]
#-# [1,]    1    6   11   16   21
#-# [2,]    2    7   12   17   22
#-# [3,]    3    8   13   18   23
#-# [4,]    4    9   14   19   24
#-# [5,]    5   10   15   20   25
#-#
#-# [[3]]
#-#  [1]  1  2  3  4  5  6  7  8  9 10

#select second element
l[[2]]

#-#      [,1] [,2] [,3] [,4] [,5]
#-# [1,]    1    6   11   16   21
#-# [2,]    2    7   12   17   22
#-# [3,]    3    8   13   18   23
#-# [4,]    4    9   14   19   24
#-# [5,]    5   10   15   20   25

#you can also use single brackets, but then the results
#will be a list with one element
l[2]

#-# [[1]]
#-#      [,1] [,2] [,3] [,4] [,5]
#-# [1,]    1    6   11   16   21
#-# [2,]    2    7   12   17   22
#-# [3,]    3    8   13   18   23
#-# [4,]    4    9   14   19   24
#-# [5,]    5   10   15   20   25

#select column two of the second element
l[[2]][,2]

#-# [1]  6  7  8  9 10

#renaming

rownames(df)

#-#  [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"

colnames(df)
```

```
#-# [1] "int_vector"  "fact_vector"

#Let's rename the second variable of df to 'Something'

colnames(df)[2] <- 'Something'
df

#-#    int_vector Something
#-# 1           1         a
#-# 2           2         b
#-# 3           3         c
#-# 4           4         d
#-# 5           5         e
#-# 6           6         f
#-# 7           7         g
#-# 8           8         h
#-# 9           9         i
#-# 10         10         j
```

2.3.3.4 Memory monitoring and management

R does all of its computation in working memory. Another term for working memory is RAM (Random Access Memory). While this makes it fast, one might quickly run out of memory. A typtical laptop computer has 8GB of RAM, and 500GB of disk space. Accessing disk space (reading and writing) is very slow compared to accessing RAM. If our computer runs out of RAM, it will try to enlist our disk space for its operations. This is called swap and should be avoided at all times. It will make our computer unresponsive. Alternatively R may crash. To avoid running out of RAM, it is important that we know how to monitor memory usage, and how to manage it. In this subsection we will discuss memory monitoring and management techniques.

Before moving on we need to clarify the difference between two common metric systems: binary and decimal. The binary system (IEC: International Elecrotechnical Commission) is specified using powers of 2. The decimal system (SI: Système International d'Unités; International System of Units) is specified using powers of 10. Table 2.2 show the difference. Kibibyte stands for kilo binary and is 1024 bytes. In contrast a kilobyte is 1000 bytes.The binary system is more commonly used in relation to RAM, whereas the decimal system is more commonly used for disk storage. That is the reason why we will be seeing a lot of KiB and MiB in this section: we are looking at RAM usage.

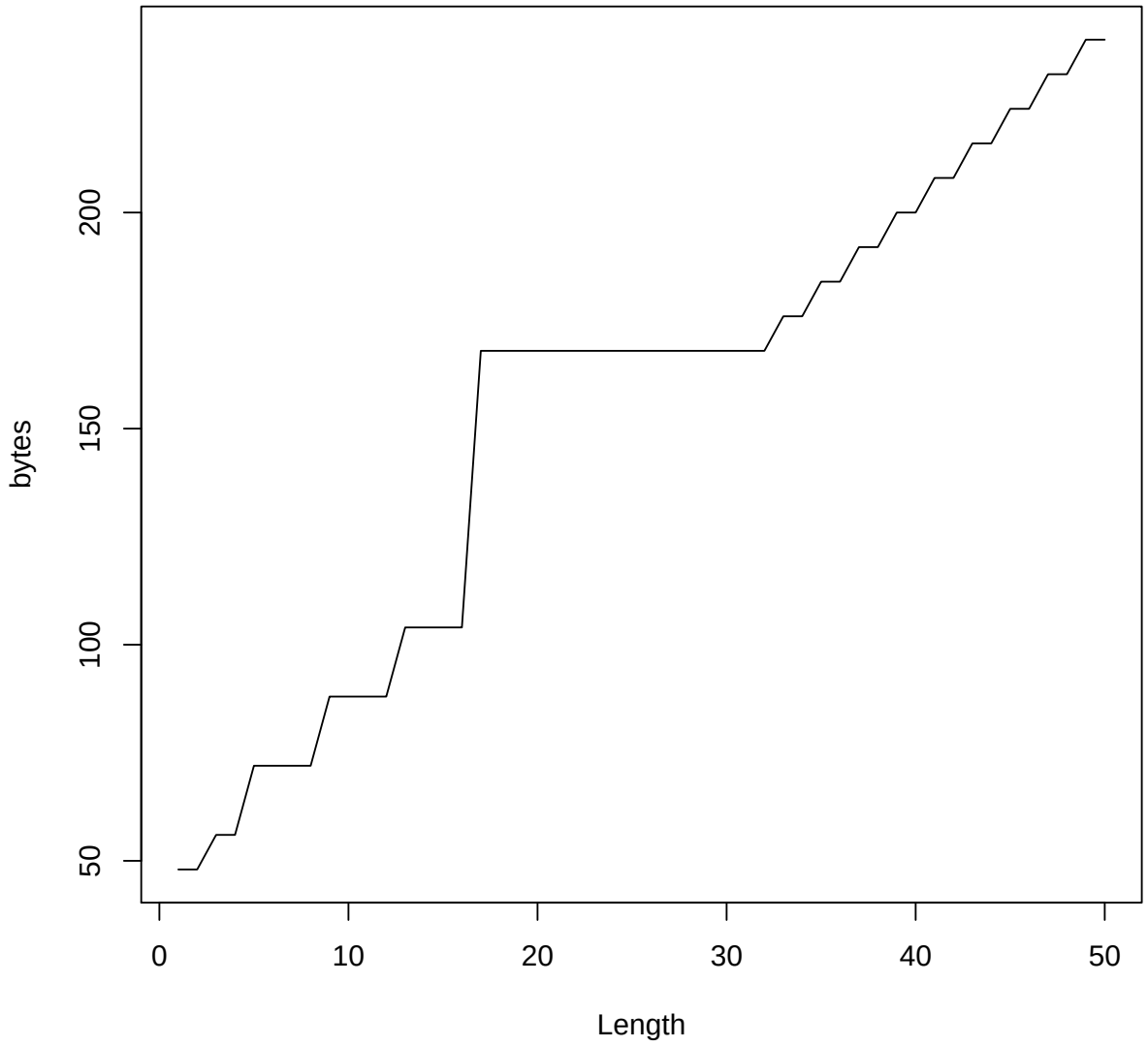
```
# Consider the following objects:
a <- 1.0
b <- 1:10

# To see all the objects that occupy the environment use
ls()

#-#  [1] "a"           "b"           "char_vector" "df"
```

Table 2.2: Prefixes used for bytes (B)

Decimal: SI			Binary: IEC		
Short	Long	Bytes	Short	Long	Bytes
kB	kilobyte	$10^3 = 1000$	KiB	kibibyte	$2^{10} = 1024$
MB	megabyte	$10^6 = 1000^2$	MiB	mebibyte	$2^{20} = 1024^2$
GB	gigabyte	$10^9 = 1000^3$	GiB	gibibyte	$2^{30} = 1024^3$
TB	terabyte	$10^{12} = 1000^4$	TiB	tebibyte	$2^{40} = 1024^4$
PB	petabyte	$10^{15} = 1000^5$	PiB	pebibyte	$2^{50} = 1024^5$
EB	exabyte	$10^{18} = 1000^6$	EiB	exbibyte	$2^{60} = 1024^6$
ZB	zettabyte	$10^{21} = 1000^7$	ZiB	zebibyte	$2^{70} = 1024^7$
YB	yottabyte	$10^{24} = 1000^8$	YiB	yobibyte	$2^{80} = 1024^8$

```
#-#  [5] "fact_vector" "int_vector"  "l"           "log_vector"
#-#  [9] "mat"         "num_vector"

# A very useful line of code is the following.
# It will remove (rm) all objects in the environment as listed by ls()
rm(list=ls())

# Let's check if we were successful:
ls()

#-# character(0)

# Yes, there are no more objects in the environment

# There are two levels at which we want to measure RAM
# consumption: micro, and macro.

# Micro level
############################
# A the micro level we are concerned with how much
# RAM is consumed by individual objects. We use the
# base R function object.size() for this task.

#Consider a vector called a and its object size
a <- 1.0
object.size(a)

#-# 48 bytes

# It might be surprising that more than the usual 8 bytes are used
# to store this numeric value in double-precision floating-point format.
# The underlying reason is that R uses 40 bytes to store meta data.
# An example of meta-data is the length, or base type of the object.
```

```r
# To see how R allocates RAM, let's grow a numeric vector from length
# 1 to length 50. The plot indicates that memory usage does not grow
# proportionally. For now do not lose too much time trying to
# understand the loop immediately below. We will cover loops in
# detail in subsequent sections.

f <- numeric()
for (i in 1:50) {
  e <- 1:i
  f[i] <- as.numeric(utils::object.size(e))
}

plot(1:50, f, type="l", xlab="Length", ylab="bytes")
```




```
# Interestingly even an object of length 0 will consume memory to store
# overhead.
(e <- numeric(0))

#-# numeric(0)

object.size(e)

#-# 40 bytes

# The reason why memory usage is disproportional is because requesting
```

```
# memory from the operating system is expensive. If R has to ask for
# memory each time something changes it would slow down R. Therefore
# R requests more memory than it needs and manages that additional memory
# itself. After 128 bytes R hands over management of vectors to the
# the operating system because the latter is very efficient in doing that.

# Using the following function we can learn which objects take up the most space.
# Let's first create some objects:
b <- matrix(runif(1000),ncol=10)
c <- list(1:10,1:100)
# This line of code lists all the objects in the working
# environment using the function ls(). It then
# applies the function object.size to each of the objects
# to get their size in bytes. If there are objects that we do not
# need anymore we can remove them. This is a very handy to free up memory.
# Again, we will explain sapply in detail in a subsequent section.
# For now, just use this line of code to see memory consumption of all
# the objects in memory.

(d <- sort(sapply(ls(all.names=TRUE),function(x){utils::object.size(get(x))}))) # in bytes

#-#            e            a            i            f
#-#           40           48           48          440
#-#            c .Random.seed            b
#-#          584         2544         8200

#Total memory used by R:
sum(d)

#-# [1] 11904

#Let's remove the object from our environment
ls()

#-# [1] "a" "b" "c" "d" "e" "f" "i"

rm(a, b, c, d)
ls()

#-# [1] "e" "f" "i"

# As far as memory management, one has to carefully reflect on
# which type of object to use.

#Consider 1000 values, each value unique
object.size(as.integer(1:1000))

#-# 4040 bytes

object.size(as.numeric(1:1000))
```

```r
#-# 8040 bytes

object.size(as.character(1:1000))

#-# 56040 bytes

object.size(as.factor(1:1000))

#-# 60400 bytes

#the last line stores 1000 labels


#Now again consider 1000 values, but there are
#only 2 unique values
object.size(as.integer(c(rep(1,500),rep(0,500))))

#-# 4040 bytes

object.size(as.numeric(c(rep(1,500),rep(0,500))))

#-# 8040 bytes

object.size(as.character(c(rep(1,500),rep(0,500))))

#-# 8136 bytes

object.size(as.factor(c(rep(1,500),rep(0,500))))

#-# 4512 bytes

#the last line stores 1k integers and 2 labels


# Let's clean up our environment

#First look at what we have in the environment:
ls()

#-# [1] "e" "f" "i"

#remove everything
rm(list=ls())

#look again:
ls()

#-# character(0)
```

```r
# Macro level
##########################
# While the micro level is concerned with how much RAM R uses (and
# objects in the environment), we now look at how much RAM is still
# available on our computer taking into account non-R processes.

# We can obtain this information from activity monitor on MacOS,
# or task manager on Windows (crtl-alt-del). We can also
# obtain this information in R as follows:

#Let's install the memuse package

if (!require("memuse")) {
  install.packages('memuse',
                   repos="https://cran.rstudio.com/",
                   quiet=TRUE)
  require('memuse')
}

#-# Loading required package: memuse

Sys.meminfo()

#-# Totalram:  16.000 GiB
#-# Freeram:    7.500 GiB

# Sys.meminfo() reports our total RAM and free RAM. We want to make sure
# we have RAM available at all times, if not our computer will freeze up.
# It is a good idea to always keep an eye on memory usage.

# Important note. Once you remove an object from R, by using rm(),
# R still hangs on to the memory that that object used,
# in case the user would need it for something else. It does
# that because returning memory to the operating system is expensive. If the
# user does not use it after a while, R will automatically return it to the
# operating system. It does that by calling the function gc(), which is
# short for garbage collection. R does this automatically in the background,
# so there is no need for you to call the gc() function.

# However, if you would want to see how Sys.meminfo() changes immediately after
# removing an object, you would run gc() so R returns the RAM back to the operating
# system. Otherwise you would have to wait until R returns it automatically after
# a while.
```

What can we conclude? Factors in R are stored as a vector of integer values with a corresponding set of labels to use when the factor is displayed. Therefore, whenever we need to choose between character and factor:

- When we have many unique values: use character
- When we have only a few unique values: use factor

We need to be careful though when working with ID's: it might be tempting to store them as integers but this will lead to incorrect imports for IDs with leading zeros:

```
as.integer('013')

#-# [1] 13

as.character('013')

#-# [1] "013"
```

2.3.3.5 Reading in data

```
# Usually we have data on our disk and we want to read it into R

# The first thing we do is set R to our working directory (where the
# data is stored). We use the setwd function. For example:
# setwd("/Users/Username/folder/"). On Windows we either use "\\" or "/",
# on Linux or Unix (Mac) we use "/" in the path. We can also
# see where our current working directory is with getwd()
# Once we have set our working directory we can download
# files as follows, or simply copy paste them into the working directory.

# Download the data:
# download.file("http://ballings.co/hidden/aCRM/data/chapter2/subscriptions.txt",
#               destfile="subscriptions.txt")

# We can then simply read the subscriptions dataset using the read.table function
# subscriptions <- read.table("subscriptions.txt",
#                             header=TRUE, sep=";")

# We can also read directly from a URL:
URL_subs <-
  "http://ballings.co/hidden/aCRM/data/chapter2/subscriptions.txt"

subscriptions <- read.table(URL_subs,
                            header=TRUE, sep=";")

#Look at the structure of the data. We see that there are
# 227 observations, and 21 variables along with the first
#couple of values of each variable
str(subscriptions, vec.len=0.5)

#-# 'data.frame': 227 obs. of  21 variables:
#-#  $ SubscriptionID   : int  1000024 ...
#-#  $ CustomerID       : int  1150045 ...
#-#  $ ProductID        : int  8 ...
#-#  $ Pattern          : int  1111110 ...
```

```
#-#  $ StartDate        : Factor w/ 58 levels "01/01/2008","01/02/2010",..: 31 ...
#-#  $ EndDate          : Factor w/ 116 levels "01/03/2011","01/04/2010",..: 57 ...
#-#  $ NbrNewspapers    : int  76 ...
#-#  $ NbrStart         : int  25 ...
#-#  $ RenewalDate      : Factor w/ 72 levels "","01/12/2007",..: 35 ...
#-#  $ PaymentType      : Factor w/ 2 levels "BT","DD": 1 ...
#-#  $ PaymentStatus    : Factor w/ 2 levels "Not Paid","Paid": 2 ...
#-#  $ PaymentDate      : Factor w/ 61 levels "","01/02/2010",..: 54 ...
#-#  $ FormulaID        : int  8552 ...
#-#  $ GrossFormulaPrice: num  79 ...
#-#  $ NetFormulaPrice  : num  79 ...
#-#  $ NetNewspaperPrice: num  1.04 ...
#-#  $ ProductDiscount  : int  0 ...
#-#  $ FormulaDiscount  : num  0 ...
#-#  $ TotalDiscount    : num  0 ...
#-#  $ TotalPrice       : num  79 ...
#-#  $ TotalCredit      : num  0 ...

#Speeding up reading in data and reducing memory usage:

# 1) The use of colClasses helps R read in data a lot faster
subscriptions <-read.table(URL_subs,
                           header=TRUE, sep=";",
                           colClasses=c("character",
                                        "character",
                                        "character",
                                        "character",
                                        "character",
                                        "character",
                                        "integer",
                                        "integer",
                                        "character",
                                        "factor",
                                        "factor",
                                        "character",
                                        "character",
                                        "numeric",
                                        "numeric",
                                        "numeric",
                                        "numeric",
                                        "numeric",
                                        "numeric",
                                        "numeric",
                                        "numeric"))

#If we are reading in a table for the first time, and we don't
# know the classes, we can use the nrow parameter in read.table.
# We can read in just a few rows of the table and then create a
#vector of classes from just the few rows.
subscriptions <- read.table(URL_subs,
                           nrow=5, header=TRUE, sep=";")
```

```
classes <- sapply(subscriptions,class)
subscriptions <- read.table(URL_subs,
                 header=TRUE, sep=";", colClasses=classes)

# 2) Another strategy to improve the speed of reading in data is
# setting the 'nrows' parameter to the exact number of rows.
#It can help a lot with memory usage. R doesn't know how many
# rows it's going to read in so it first makes a guess, and then
# when it runs out of room it allocates more memory. The constant
# allocations can take a lot of time, and if R overestimates the
#amount of memory it needs, our computer might run out of memory.

#Of course, we may not know how many rows our table has.
#The easiest way to find out is to look at the data
# using a text editor. If we're on Unix (Mac) we can also
#use the 'wc' command. So if we run 'wc datafile.txt' in Unix,
#then it will report to you the number of lines in the
#file (the first number). We can then pass this number to
#the 'nrows' argument of 'read.table()'.

#If we can't use the 'wc' (wordcount) command for some reason,
# but we know that there are definitely less than N rows,
# then we can specify 'nrows = N' and things will still be okay.
#A small overestimate for 'nrows' is better than not setting nrows.

#3) A third strategy to improve read speed is to set 'comment.char'
# in read.table. If our file has no comments in it (e.g., lines
# starting with '#'), then setting comment.char = "" will
#sometimes make 'read.table()' run faster.
```

2.3.3.6 Data exploration

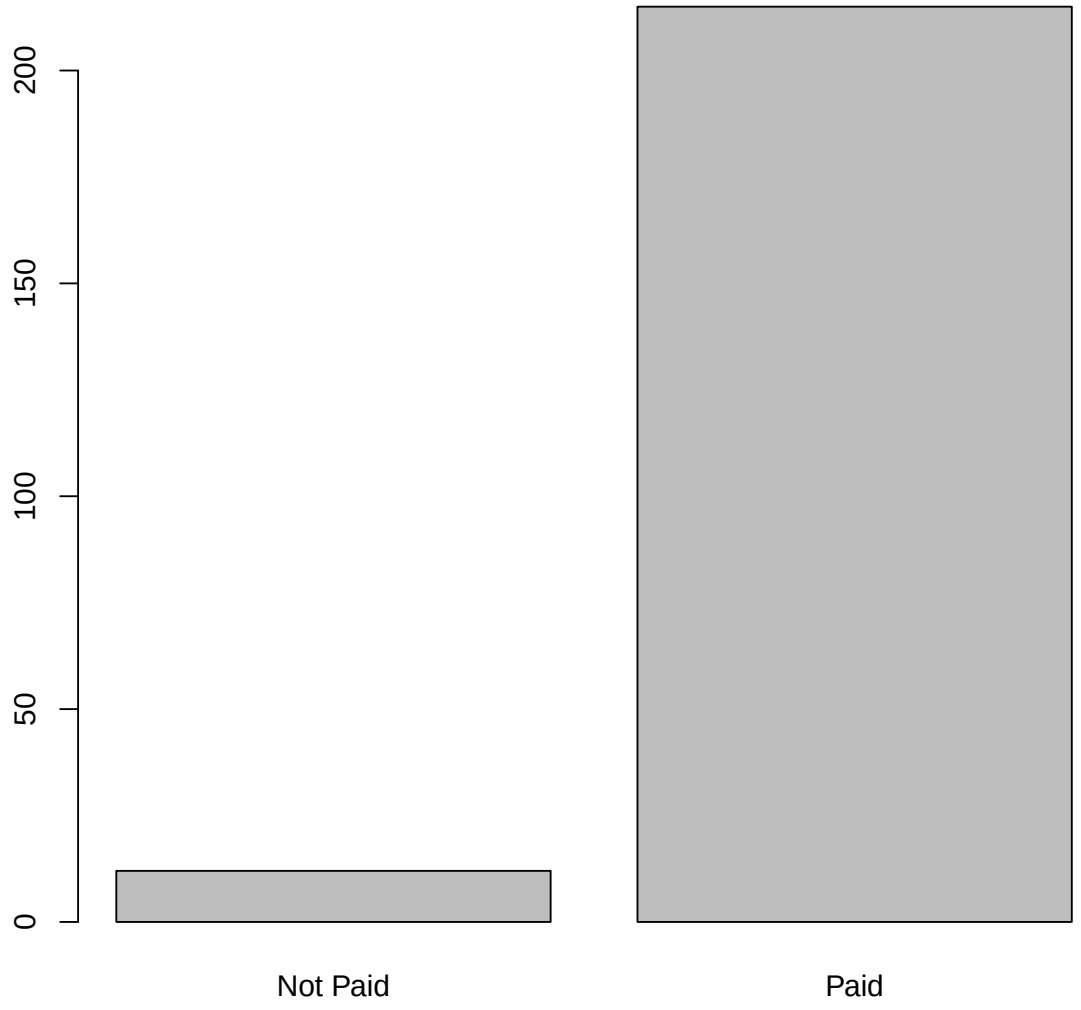
```
URL_subs <-
  "http://ballings.co/hidden/aCRM/data/chapter2/subscriptions.txt"

subscriptions <- read.table(URL_subs,
                            header=TRUE, sep=";")

#What is the class of the object?
class(subscriptions)

#-# [1] "data.frame"

#What are its dimensions? The first number is the number of rows,
#the second number is the number of columns.
dim(subscriptions)

#-# [1] 227  21
```

```
nrow(subscriptions) #same as dim(subscriptions)[1]

#-# [1] 227

ncol(subscriptions) #same as dim(subscriptions)[2]

#-# [1] 21

#Look at the structure of the object
str(subscriptions,vec.len=0.5)

#-# 'data.frame': 227 obs. of  21 variables:
#-#  $ SubscriptionID   : int  1000024 ...
#-#  $ CustomerID       : int  1150045 ...
#-#  $ ProductID        : int  8 ...
#-#  $ Pattern          : int  1111110 ...
#-#  $ StartDate        : Factor w/ 58 levels "01/01/2008","01/02/2010",..: 31 ...
#-#  $ EndDate          : Factor w/ 116 levels "01/03/2011","01/04/2010",..: 57 ...
#-#  $ NbrNewspapers    : int  76 ...
#-#  $ NbrStart         : int  25 ...
#-#  $ RenewalDate      : Factor w/ 72 levels "","01/12/2007",..: 35 ...
#-#  $ PaymentType      : Factor w/ 2 levels "BT","DD": 1 ...
#-#  $ PaymentStatus    : Factor w/ 2 levels "Not Paid","Paid": 2 ...
#-#  $ PaymentDate      : Factor w/ 61 levels "","01/02/2010",..: 54 ...
#-#  $ FormulaID        : int  8552 ...
#-#  $ GrossFormulaPrice: num  79 ...
#-#  $ NetFormulaPrice  : num  79 ...
#-#  $ NetNewspaperPrice: num  1.04 ...
#-#  $ ProductDiscount  : int  0 ...
#-#  $ FormulaDiscount  : num  0 ...
#-#  $ TotalDiscount    : num  0 ...
#-#  $ TotalPrice       : num  79 ...
#-#  $ TotalCredit      : num  0 ...

#Display the first couple of rows
head(subscriptions)

#-#   SubscriptionID CustomerID ProductID Pattern  StartDate
#-# 1        1000024    1150045         8 1111110 18/01/2010
#-# 2        1000082    1150046         8 1111110 14/01/2010
#-# 3         100012      89085         8 1111110 01/01/2008
#-# 4         100013      89085         8 1111110 02/01/2007
#-# 5         100016      89087         8 1111110 01/01/2008
#-# 6         100017      89087         8 1111110 02/01/2007
#-#      EndDate NbrNewspapers NbrStart RenewalDate PaymentType
#-# 1 17/04/2010            76       25  17/03/2010          BT
#-# 2 12/03/2010            50       25  03/03/2010          BT
#-# 3 30/12/2008           304       10  21/11/2008          BT
#-# 4 31/12/2007           304       25  01/12/2007          BT
#-# 5 30/12/2008           304       10  21/11/2008          BT
#-# 6 31/12/2007           304       25  01/12/2007          BT
```

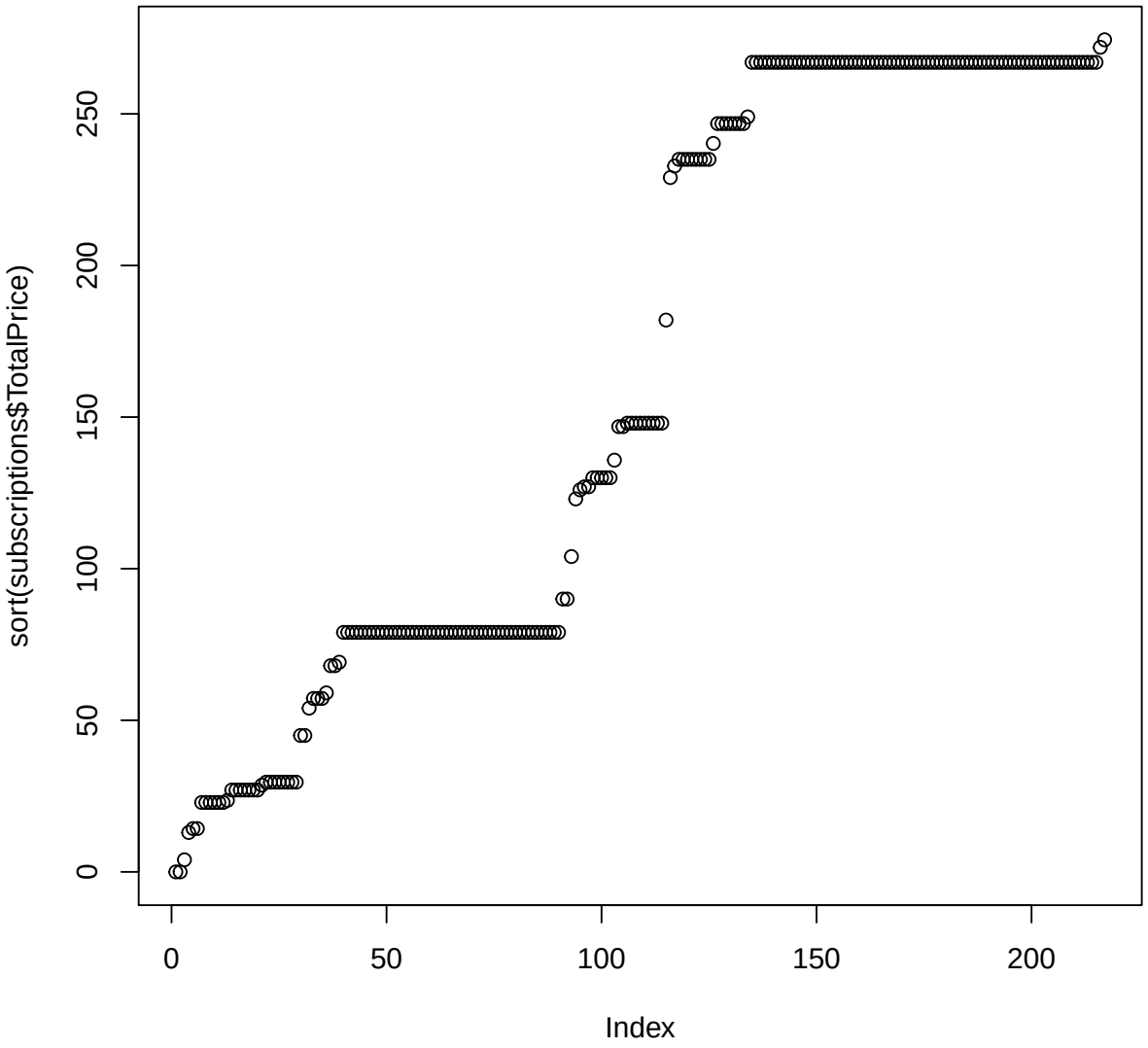
```
#-#   PaymentStatus PaymentDate FormulaID GrossFormulaPrice
#-# 1          Paid  27/01/2010      8552           79.0000
#-# 2          Paid  11/02/2010     10017           51.9737
#-# 3          Paid  12/12/2007       875          249.0000
#-# 4          Paid  28/12/2006       874          235.0000
#-# 5          Paid  12/12/2007       875          249.0000
#-# 6          Paid  13/12/2006       874          235.0000
#-#   NetFormulaPrice NetNewspaperPrice ProductDiscount
#-# 1            79.0          1.039474               0
#-# 2            29.6          0.592000               0
#-# 3           249.0          0.811842               0
#-# 4           235.0          0.773026               0
#-# 5           249.0          0.811842               0
#-# 6           235.0          0.773026               0
#-#   FormulaDiscount TotalDiscount TotalPrice TotalCredit
#-# 1          0.0000        0.0000       79.0         0.0
#-# 2         22.3737       22.3737       29.6         0.0
#-# 3          0.0000        0.0000      246.8        -2.2
#-# 4          0.0000        0.0000      235.0         0.0
#-# 5          0.0000        0.0000      246.8        -2.2
#-# 6          0.0000        0.0000      235.0         0.0

#Display the last couple of rows
tail(subscriptions)

#-#     SubscriptionID CustomerID ProductID Pattern  StartDate
#-# 222         101365      89989         8 1111110 14/05/2007
#-# 223         101366      89989         8 1111110 13/11/2006
#-# 224        1013721     159085         8 1111110 02/03/2010
#-# 225        1013821     449595         8 1111110 02/03/2010
#-# 226        1013860     659269         7 1111110 02/03/2010
#-# 227        1013944     760585         6 1111110 02/03/2010
#-#        EndDate NbrNewspapers NbrStart RenewalDate
#-# 222 12/11/2007           152       25  13/10/2007
#-# 223 12/05/2007           152       15
#-# 224 01/03/2011           304       10  31/01/2011
#-# 225 01/03/2011           304       10  31/01/2011
#-# 226 01/03/2011           304       10  31/01/2011
#-# 227 02/03/2010           304       10
#-#     PaymentType PaymentStatus PaymentDate FormulaID
#-# 222          BT          Paid  31/05/2007       869
#-# 223          BT          Paid  07/03/2007       153
#-# 224          BT          Paid  16/02/2010      8637
#-# 225          BT          Paid  03/03/2010      8637
#-# 226          BT          Paid  01/03/2010      8640
#-# 227          BT      Not Paid                  8896
#-#     GrossFormulaPrice NetFormulaPrice NetNewspaperPrice
#-# 222               127             127          0.026316
#-# 223               123             123          0.809211
#-# 224               267             267          0.878289
#-# 225               267             267          0.878289
```

```
#-# 226               267             267          0.878289
#-# 227                NA              NA                NA
#-#     ProductDiscount FormulaDiscount TotalDiscount
#-# 222               0               0             0
#-# 223               0               0             0
#-# 224               0               0             0
#-# 225               0               0             0
#-# 226               0               0             0
#-# 227              NA              NA            NA
#-#     TotalPrice TotalCredit
#-# 222          4        -123
#-# 223        123           0
#-# 224        267           0
#-# 225        267           0
#-# 226        267           0
#-# 227         NA          NA

#How long is a vector? This is the same as nrow(subscriptions)
length(subscriptions$PaymentStatus)

#-# [1] 227

#What are the column names?
colnames(subscriptions)

#-#  [1] "SubscriptionID"    "CustomerID"
#-#  [3] "ProductID"         "Pattern"
#-#  [5] "StartDate"         "EndDate"
#-#  [7] "NbrNewspapers"     "NbrStart"
#-#  [9] "RenewalDate"       "PaymentType"
#-# [11] "PaymentStatus"     "PaymentDate"
#-# [13] "FormulaID"         "GrossFormulaPrice"
#-# [15] "NetFormulaPrice"   "NetNewspaperPrice"
#-# [17] "ProductDiscount"   "FormulaDiscount"
#-# [19] "TotalDiscount"     "TotalPrice"
#-# [21] "TotalCredit"

#Look at a random subset of the data.
subscriptions[sample(1:nrow(subscriptions),5),]

#-#     SubscriptionID CustomerID ProductID Pattern  StartDate
#-# 188        1011376     722525         6      10 28/02/2010
#-# 7          1000372     325943         2 1111110 13/02/2010
#-# 25         1001450    1150113         8 1111110 15/01/2010
#-# 163        1009920     455810         2 1111110 26/02/2010
#-# 189        1011473     732899         2 1111110 29/01/2010
#-#        EndDate NbrNewspapers NbrStart RenewalDate
#-# 188 04/09/2010            26        4  05/08/2010
#-# 7   12/02/2011           304       10  13/01/2011
#-# 25  12/02/2010            25       25  04/03/2010
#-# 163 03/03/2011           304       10  19/08/2010
```

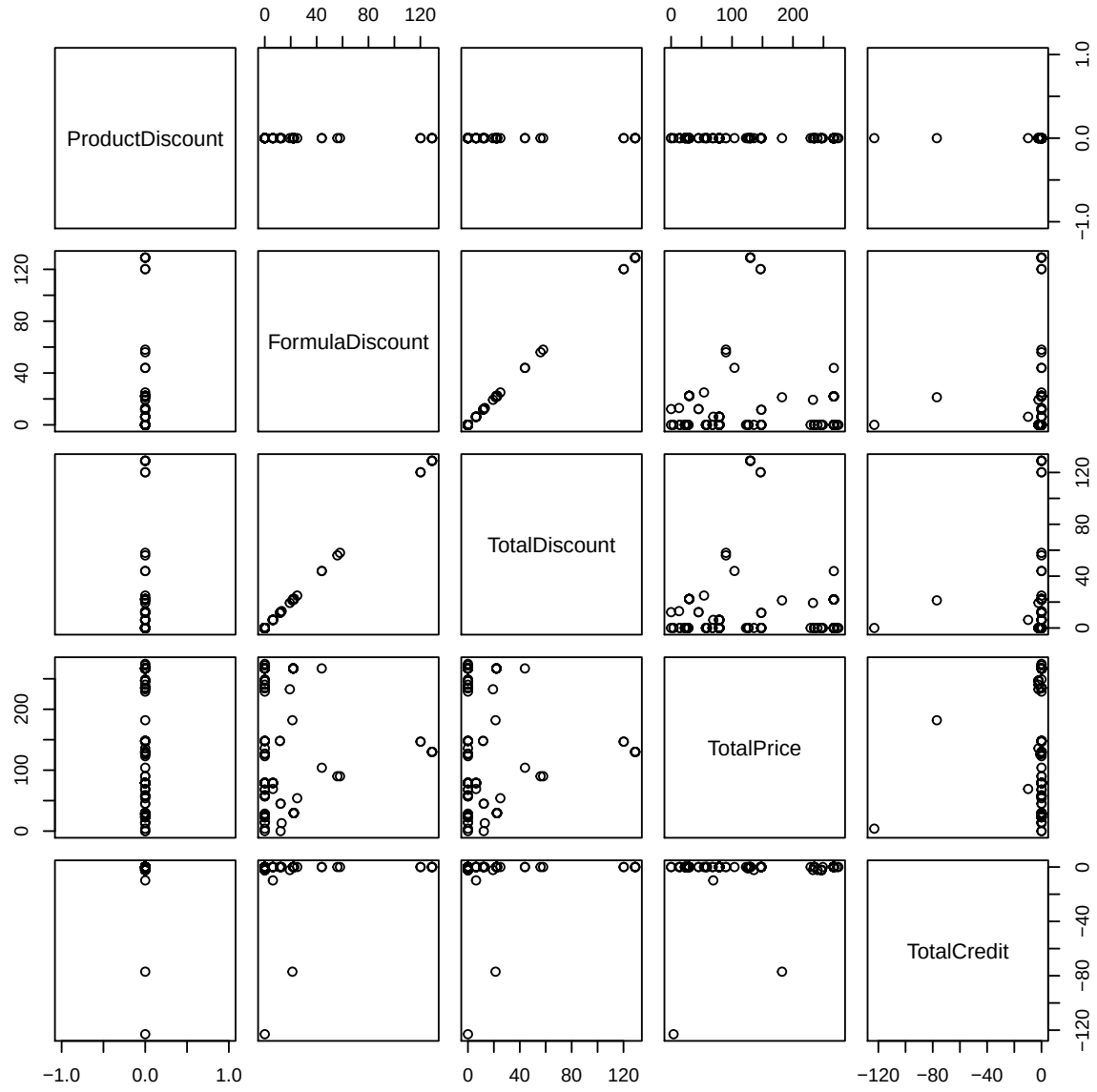
```
#-# 189 06/05/2010            82       25  06/04/2010
#-#     PaymentType PaymentStatus PaymentDate FormulaID
#-# 188          BT          Paid  18/02/2010      1932
#-# 7            BT          Paid  27/01/2010      8640
#-# 25           BT          Paid  15/02/2010      6299
#-# 163          BT          Paid  02/03/2010      8640
#-# 189          BT          Paid  15/02/2010      9755
#-#     GrossFormulaPrice NetFormulaPrice NetNewspaperPrice
#-# 188          28.60000            28.6          1.100000
#-# 7           267.00000           267.0          0.878289
#-# 25           26.00000            13.0          0.520000
#-# 163         267.00000           267.0          0.878289
#-# 189          85.23687            79.0          0.963415
#-#     ProductDiscount FormulaDiscount TotalDiscount
#-# 188               0        0.000000      0.000000
#-# 7                 0        0.000000      0.000000
#-# 25                0       13.000000     13.000000
#-# 163               0        0.000000      0.000000
#-# 189               0        6.236868      6.236868
#-#     TotalPrice TotalCredit
#-# 188       28.6           0
#-# 7        267.0           0
#-# 25        13.0           0
#-# 163      267.0           0
#-# 189       79.0           0

#Frequencies
table(subscriptions$PaymentStatus)

#-#
#-# Not Paid     Paid
#-#       12      215

#Many functions are object-oriented: the same function
#reacts differently to different data types, this is called method dispatching

plot(subscriptions$PaymentStatus)
```




```
plot(sort(subscriptions$TotalPrice))
```

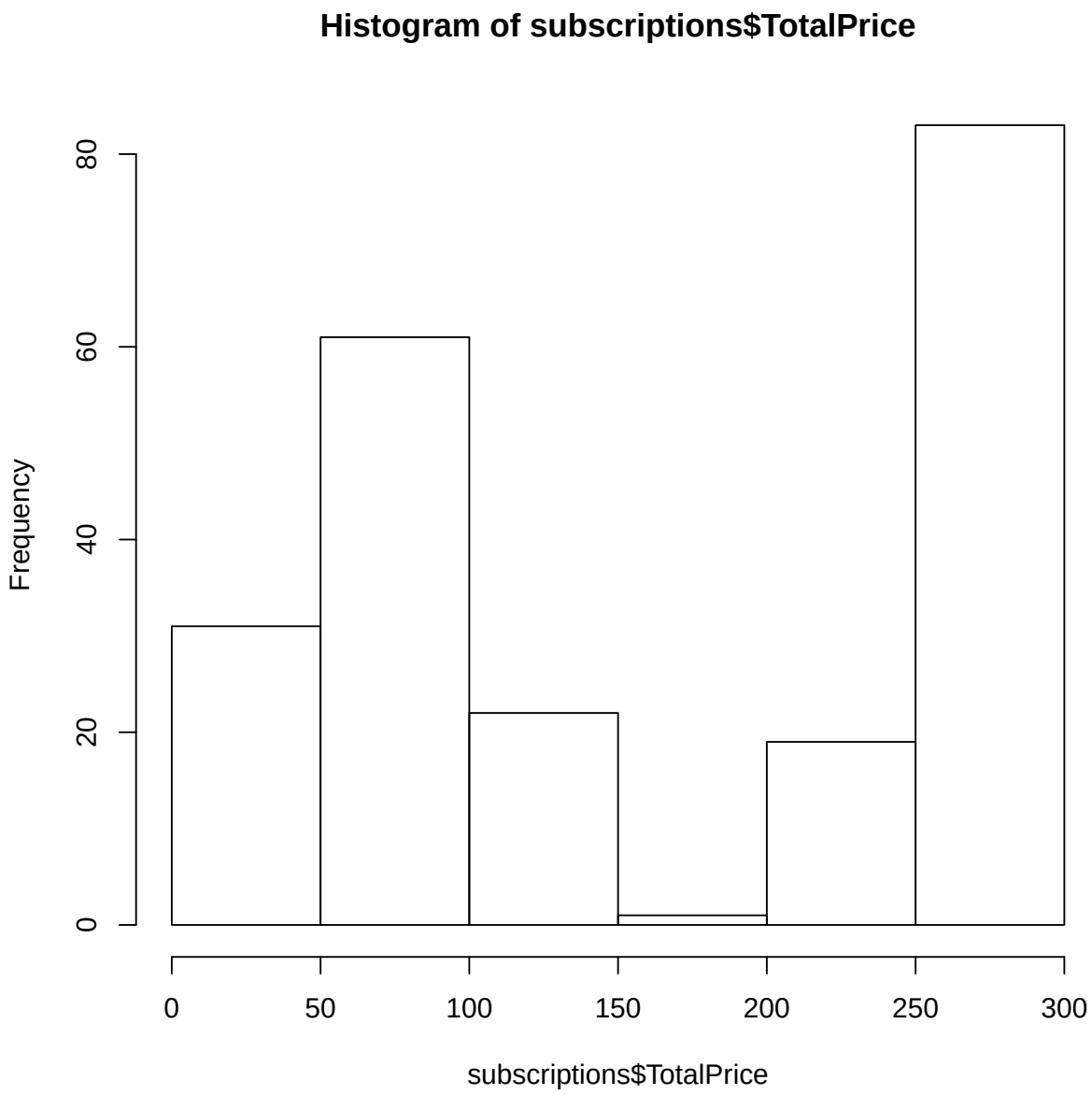



```
pairs(subscriptions[,17:21]) # a scatterplot matrix of variables in the data
```




```
#summarize a factor
summary(subscriptions$PaymentStatus)

#-# Not Paid     Paid
#-#       12      215

#summarize a numeric variable
summary(subscriptions$TotalPrice)

#-#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
#-#     0.0    79.0   148.0   163.1   267.0   274.4      10
```

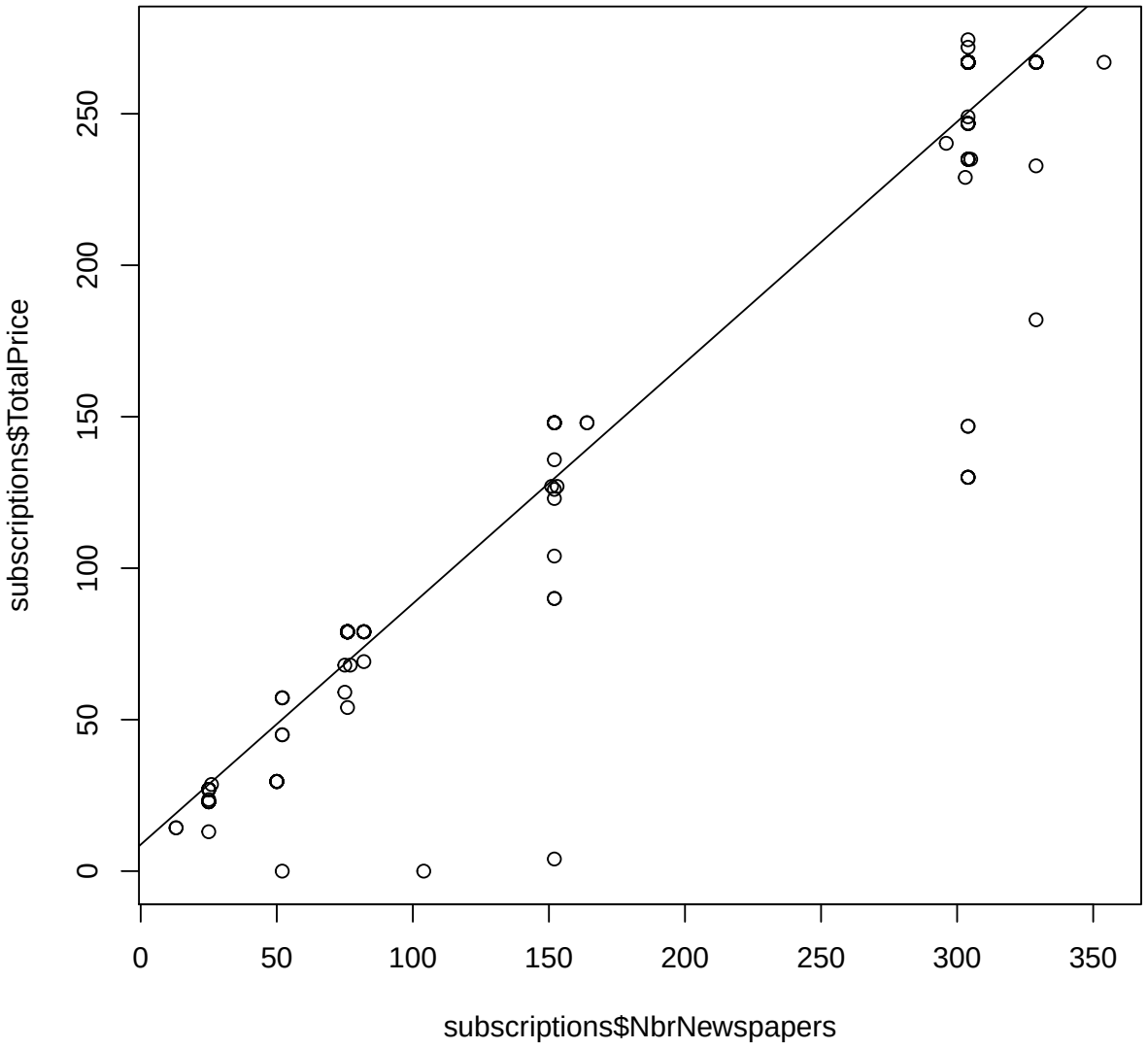
```r
#A histogram of TotalPrice
hist(subscriptions$TotalPrice)
```




```r
#Mean
mean(subscriptions$TotalPrice, na.rm=TRUE)

#-# [1] 163.0524

#Variance
var(subscriptions$TotalPrice, na.rm=TRUE)
```

```
#-# [1] 9654.953

#Standard deviation
sd(subscriptions$TotalPrice, na.rm=TRUE)

#-# [1] 98.25962

#Which values are not missing in TotalPrice?
!is.na(subscriptions$TotalPrice)

#-#   [1]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#-#  [10] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#-#  [19]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#-#  [28]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#-#  [37]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#-#  [46]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#-#  [55]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#-#  [64]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE  TRUE
#-#  [73]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#-#  [82]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#-#  [91]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#-# [100]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#-# [109]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#-# [118]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#-# [127]  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#-# [136] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#-# [145]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#-# [154]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#-# [163]  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#-# [172]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#-# [181]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#-# [190]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#-# [199] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#-# [208]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#-# [217]  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
#-# [226]  TRUE FALSE

#Correlation between two variables
cor(subscriptions$TotalPrice[!is.na(subscriptions$TotalPrice)],
    subscriptions$NbrNewspapers[!is.na(subscriptions$TotalPrice)])

#-# [1] 0.9588973

#this is easier:
cor(subscriptions$TotalPrice,
    subscriptions$NbrNewspapers, use="complete.obs")

#-# [1] 0.9588973

#Range of a variable
range(subscriptions$TotalPrice[!is.na(subscriptions$TotalPrice)])
```

```
#-# [1]   0.00 274.44

#Minimum
min(subscriptions$TotalPrice, na.rm=TRUE)

#-# [1] 0

#Maximum
max(subscriptions$TotalPrice, na.rm=TRUE)

#-# [1] 274.44

#Is this a numeric vector?
is.numeric(subscriptions$TotalPrice)

#-# [1] TRUE

is.numeric(subscriptions$CustomerID)

#-# [1] TRUE

#Scatterplot with regression line
plot(subscriptions$NbrNewspapers,subscriptions$TotalPrice)
abline(lm(TotalPrice ~ NbrNewspapers, subscriptions))
```



2.3.3.7 Installing and loading packages

```
#-#
#-#   There are binary versions available but the source
#-#   versions are later:
#-#             binary source needs_compilation
#-# curl           2.3    2.4              TRUE
#-# sourcetools  0.1.5  0.1.6              TRUE
#-#
#-#
```

```
#-# The downloaded binary packages are in
#-#  /var/folders/5g/s33ddpz92_bcr3lz_k2wzjbc0000gn/T//RtmpqmydAn/downloaded_packages

#-# installing the source packages 'curl', 'sourcetools'

#To check the version of R we can do:
getRversion()

#-# [1] '3.3.2'

#Installing packages can be done using 'install.packages'.
#For example if we want to install package 'randomForest' we would use:

install.packages('randomForest',
                 repos="https://cran.rstudio.com/",
                 quiet=TRUE)

#The repos and quiet parameters are optional.

# To check the version of the package:
packageVersion("randomForest")

#-# [1] '4.6.12'

# To obtain a description:
packageDescription("randomForest")

#-# Package: randomForest
#-# Title: Breiman and Cutler's Random Forests for
#-#       Classification and Regression
#-# Version: 4.6-12
#-# Date: 2015-10-06
#-# Depends: R (>= 2.5.0), stats
#-# Suggests: RColorBrewer, MASS
#-# Author: Fortran original by Leo Breiman and Adele
#-#       Cutler, R port by Andy Liaw and Matthew Wiener.
#-# Description: Classification and regression based on a
#-#       forest of trees using random inputs.
#-# Maintainer: Andy Liaw <andy_liaw@merck.com>
#-# License: GPL (>= 2)
#-# URL:
#-#       https://www.stat.berkeley.edu/~breiman/RandomForests/
#-# NeedsCompilation: yes
#-# Packaged: 2015-10-06 18:28:23 UTC; Liawand
#-# Repository: CRAN
#-# Date/Publication: 2015-10-07 08:38:34
#-# Built: R 3.3.0; x86_64-apple-darwin13.4.0; 2016-05-04
#-#       14:07:23 UTC; unix
#-#
#-# -- File: /Library/Frameworks/R.framework/Versions/3.3/Resources/library/randomForest/Meta/package.rds
```

```r
#Loading the package (making it active) goes as follows:
library(randomForest)

#-# randomForest 4.6-12
#-# Type rfNews() to see new features/changes/bug fixes.

#To get more information while loading the package
# use the help parameter:
library(help=randomForest)

# If we try to load an uninstalled package we get an error:
library(azYYmihza)

#-# Error in library(azYYmihza): there is no package called 'azYYmihza'

# If we want to store in a variable if we loaded the
# package successfully we can do (note the parameter
# logical.return):
loaded <- library(randomForest, logical.return=TRUE)
loaded

#-# [1] TRUE

# Note that the error is replaced by a warning.

#If we try to load an uninstalled package (azAoef) we get FALSE:
loaded_lib <- library(azAoef, logical.return=TRUE)

#-# Warning in library(azAoef, logical.return = TRUE): there is no package called 'azAoef'

loaded_lib

#-# [1] FALSE

# require() is equivalent to library with logical.return=TRUE
loaded_req <- require(azAoef)

#-# Loading required package: azAoef
#-# Warning in library(package, lib.loc = lib.loc, character.only = TRUE, logical.return =
TRUE, : there is no package called 'azAoef'

loaded_req

#-# [1] FALSE

#Unloading a package goes as follows. You might want to
# do this to avoid conflicts:
unloadNamespace("randomForest")

#Get a list of all the loaded packages:
search()
```

```
#-#  [1] ".GlobalEnv"        "package:memuse"
#-#  [3] "package:knitr"     "package:stats"
#-#  [5] "package:graphics"  "package:grDevices"
#-#  [7] "package:utils"     "package:datasets"
#-#  [9] "package:methods"   "Autoloads"
#-# [11] "package:base"

# It is always a good idea to first check if a package is
# already installed to avoid losing time reinstalling it.
# The function "require" loads the package.
# In addition it returns FALSE if the package is not installed,
# otherwise it returns TRUE. An exclamation point negates the value.
# It means NOT. For example:
!FALSE

#-# [1] TRUE

# An "if" statement is only executed if its value is TRUE. Hence,
# if the package "randomForest" is not installed, we will get
# a FALSE, which is tranformed into a TRUE, which in turn will
# trigger the execution of the statements in the "if" statement:
# install and load the package.
# If the package is installed, it will simply be loaded by the
# first require. That first require will also return a TRUE, which
# is transformed into a FALSE, which in turn does not trigger the
# statements in the "if" statement.

if (!require("randomForest")) {
  install.packages('randomForest',
                   repos="https://cran.rstudio.com/",
                   quiet=TRUE)
  require('randomForest')
}

#-# Loading required package: randomForest
#-# randomForest 4.6-12
#-# Type rfNews() to see new features/changes/bug fixes.

# Name clashes can happen when installing packages
# If two packages have a function with the same function name
# the function of the package that was loaded last
# will be accessible on the search path. While loading
# the package a warning will be printed to say that
# a function is masked. Note that we need to set quiet=FALSE
# in install.packages.

# This can lead to surprising error messages. One time
# the function works, and then it stops working. This is because
# after loading a new package that masks our function,
# we are now using another function (with the same name).
# Let us demonstrate the problem:
```

```
# Let's load a package called AUC and use the function auc.

if (!require("AUC")) {
  install.packages('AUC',
                   repos="https://cran.rstudio.com/",
                   quiet=FALSE)
  require('AUC')
}

#-# Loading required package: AUC
#-# AUC 0.3.0
#-# Type AUCNews() to see the change log and ?AUC to get an overview.

data(churn)
auc(roc(churn$predictions,churn$labels))

#-# [1] 0.8439201

# Now let's load a package called glmnet.

if (!require("glmnet")) {
  install.packages('glmnet',
                   repos="https://cran.rstudio.com/",
                   quiet=FALSE)
  require('glmnet')
}

#-# Loading required package: glmnet
#-# Loading required package: Matrix
#-# Loading required package: foreach
#-# Loaded glmnet 2.0-5
#-#
#-# Attaching package: 'glmnet'
#-# The following object is masked from 'package:AUC':
#-#
#-#     auc

#It is saying that the auc function of the AUC package is masked.

# Let's see how this impacts us:

auc(roc(churn$predictions,churn$labels))

#-# Error in rank(prob): argument "prob" is missing, with no default

# The exact same code now throws us an error about a missing
# argument. This is because we are now using the auc
# function of the glmnet package and we want to use the auc
# function of the AUC package. We can either unload the glmnet
# package or simply put 'packagename::' in front of the function.
# This will tell R that we need the function of that specific package.
# In our case that works out as follows:

AUC::auc(roc(churn$predictions,churn$labels))
```

```
#-# [1] 0.8439201

# From time to time it is good idea to make sure
# you are running the latest versions of all packages
# You can do that with the following function:
update.packages(ask=FALSE, repos="https://cran.rstudio.com/")
```

2.3.3.8 Missing values

```
if (!require("imputeMissings")) {
  install.packages('imputeMissings',
                   repos="https://cran.rstudio.com/",
                   quiet=FALSE)
  require('imputeMissings')
}

#-# Loading required package: imputeMissings

#Compute the values on a training dataset and
#impute them on new data.
#This is very convenient in predictive contexts. For example:

#define training data
(train <- data.frame(v_int=as.integer(c(3,3,2,5,1,2,4,6)),
                     v_num=as.numeric(c(4.1,NA,12.2,11,3.4,1.6,3.3,5.5)),
                     v_fact=as.factor(c('one','two',NA,'two','two','one','two','two')),
                     stringsAsFactors = FALSE))

#-#   v_int v_num v_fact
#-# 1     3   4.1    one
#-# 2     3    NA    two
#-# 3     2  12.2   <NA>
#-# 4     5  11.0    two
#-# 5     1   3.4    two
#-# 6     2   1.6    one
#-# 7     4   3.3    two
#-# 8     6   5.5    two

#Compute values on train data
#randomForest method
values <- compute(train, method="randomForest")
#median/mode method
(values2 <- compute(train))

#-# $v_int
#-# [1] 3
#-#
#-# $v_num
#-# [1] 4.1
```

```
#-#
#-# $v_fact
#-# [1] "two"

#define new data
(newdata <- data.frame(v_int=as.integer(c(1,1,2,NA)),
                       v_num=as.numeric(c(1.1,NA,2.2,NA)),
                       v_fact=as.factor(c('one','one','one',NA)),
                       stringsAsFactors = FALSE))

#-#   v_int v_num v_fact
#-# 1     1   1.1    one
#-# 2     1    NA    one
#-# 3     2   2.2    one
#-# 4    NA    NA   <NA>

#locate the NA's
is.na(newdata)

#-#      v_int v_num v_fact
#-# [1,] FALSE FALSE  FALSE
#-# [2,] FALSE  TRUE  FALSE
#-# [3,] FALSE FALSE  FALSE
#-# [4,]  TRUE  TRUE   TRUE

#how many missings per variable?
colSums(is.na(newdata))

#-#  v_int  v_num v_fact
#-#      1      2      1

#Impute on newdata
impute(newdata,object=values) #using randomForest values

#-#      v_int    v_num v_fact
#-# 1 1.000000 1.100000    one
#-# 2 1.000000 4.062343    one
#-# 3 2.000000 2.200000    one
#-# 4 2.564879 4.062343    one

impute(newdata,object=values2) #using median/mode values

#-#   v_int v_num v_fact
#-# 1     1   1.1    one
#-# 2     1   4.1    one
#-# 3     2   2.2    one
#-# 4     3   4.1    two

#One can also impute directly in newdata without the compute step
impute(newdata)
```

```
#-#   v_int v_num v_fact
#-# 1     1  1.10    one
#-# 2     1  1.65    one
#-# 3     2  2.20    one
#-# 4     1  1.65    one

# In data mining it may be useful to have indicators
# for the variables imputations. These indicators may
# be predictive. To create these indicators, use flag=TRUE

#Flag parameter
impute(newdata,flag=TRUE)

#-#   v_int v_num v_fact v_int_flag v_num_flag v_fact_flag
#-# 1     1  1.10    one          0          0           0
#-# 2     1  1.65    one          0          1           0
#-# 3     2  2.20    one          0          0           0
#-# 4     1  1.65    one          1          1           1
```

2.3.3.9 Getting help

If we don't know how to do something Google is where we should look first. If we know the name of the function but need more information on how to use it, or its parameters we use R's built-in help system as follows: `help(function)` or `?function`. If no help is provided we can also use `help.search("function")` or `??function` to dig deeper.

2.3.4 Problems

2.4 Data Preparation

The data preparation stage consists in computing variables and aggregating and merging tables. The final objective of this phase is to create a basetable. A basetable contains information from multiple tables and is ready to apply algorithms to. In general, the more variables we have computed, the higher the likelihood that the model is of high predictive performance. In complex projects, or in projects that are executed by teams, it is always a good idea to create a code flowchart. This also forces us to think about all the tasks that need to be completed to compute a basetable. A code flowchart can form the basis for creating a task division among team members. In addition it is very handy for code maintenance.

2.4.1 Time window

In order to be able to predict the future we need to learn from the past. We first estimate a model that uses data from the far past to predict the recent past. Figure 2.4 details this methodology. In the model estimation phase we first cut up all the data in two periods: the independent or

predictor period (the period between t1 and t2), and the dependent or response period (the period between t3 and t4). The very small period between t2 and t3 is called the operational period and should only be a couple of days. The operational period comprises the time that is needed to deploy the model. To compute predictors we only use information from the independent period. To compute the dependent variable (e.g., churn, up-sell) we only use data from the dependent period. This process is displayed in the first couple of steps of the code flowchart in Section 2.4.2. Going back to Figure 2.4, once we have estimated our model we can shift our time window forward in time to extrapolate into the future. It is important that the length of the independent period should be identical in both model estimation and model deployment.

Note the difference in Figure 2.4 and Figure 2.5. In Figure 2.4 t2=t4=now. This typically happens when the runtime to estimate the model is negligible. In those cases we would always want to estimate the model on the most recent data available. In Figure 2.5 t=2, and t4 is farther back in time. This typically happens when it takes some time to estimate the model and there is no time to re-estimate it when it's time to deploy the model. In those cases the runtime is the difference between 'now' and t4 in the estimation phase. In sum, in deployment t2 always equals 'now', and in estimation, t4 equals 'now' only the optimal case runtime is negligible (otherwise t4 < 'now').

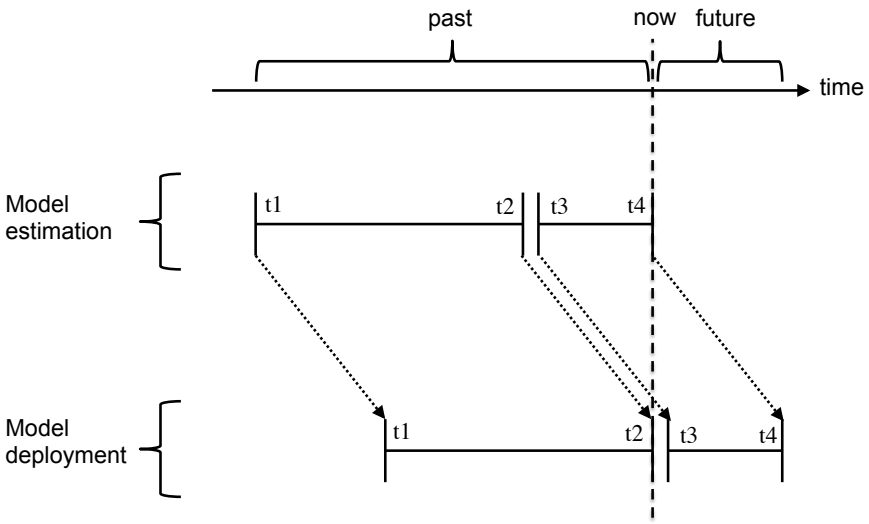


Figure 2.4: Example of a time window when the model is estimated and deployed immediately after estimation

2.4.2 Code flowchart

A code flowchart summarizes all the steps that lead to the basetable. Try to fit everything on one page and put actions in more blocks when necessary. Provide as much detail as possible but keep it comprehensible. Figure 2.6 contains an example for a single table.

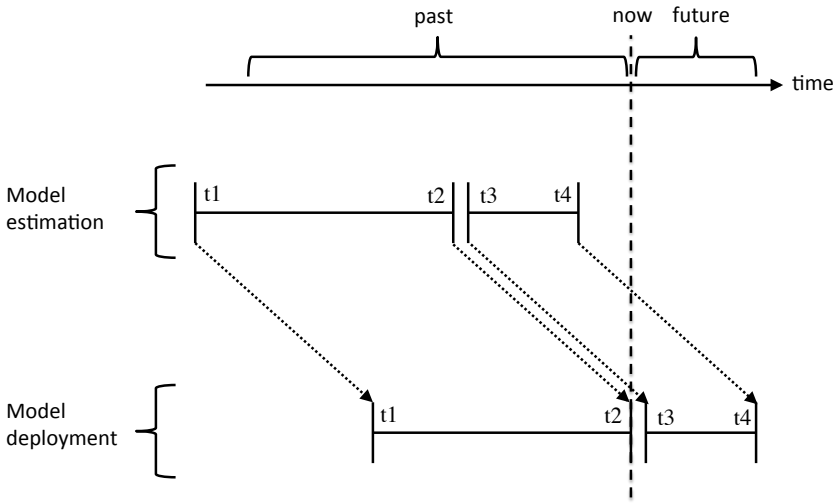


Figure 2.5: Example of a time window when the model is estimated earlier in time

2.4.3 Code Overview

This section will provide code examples of several important tasks that we will need to perform when preparing our basetable for analysis: creating dummy variables, summarizing (aggregating) data, merging (joining) data, applying functions to data frames, working with dates, creating functions, conditional processing, creating loops, and timing code.

All code in this section is available at:
ballings.co/hidden/aCRM/code/chapter2/DataPrep.R

2.4.3.1 Operators

```
# Consider the following two variables:
(x <- 1:5)

#-# [1] 1 2 3 4 5

(y <- 3)

#-# [1] 3

################################
#Relational operators
?"<"
x < y
```

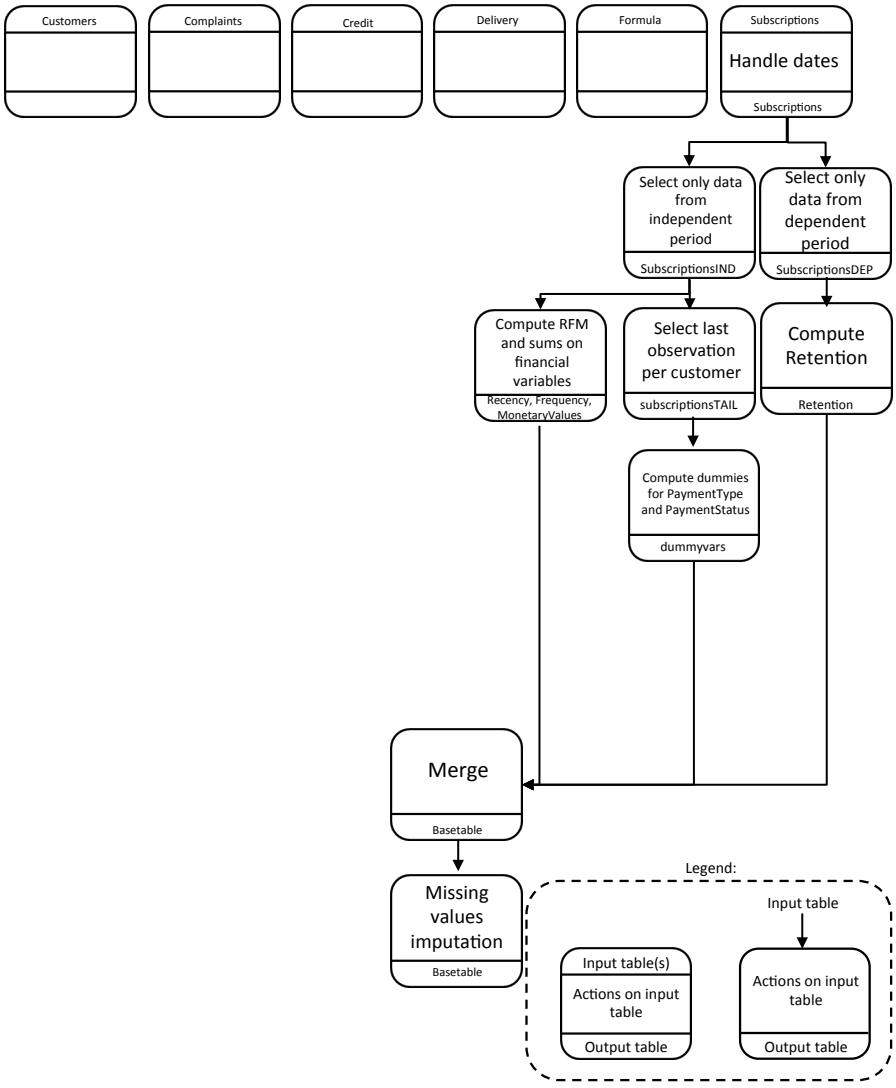


Figure 2.6: Example of a code flowchart

```
#-# [1]  TRUE  TRUE FALSE FALSE FALSE

x <= y

#-# [1]  TRUE  TRUE  TRUE FALSE FALSE

x > y

#-# [1] FALSE FALSE FALSE  TRUE  TRUE

x >= y

#-# [1] FALSE FALSE  TRUE  TRUE  TRUE

x == y # note the double =

#-# [1] FALSE FALSE  TRUE FALSE FALSE

x != y

#-# [1]  TRUE  TRUE FALSE  TRUE  TRUE

x %in% y #which elements of x appear in y?

#-# [1] FALSE FALSE  TRUE FALSE FALSE

y %in% x #which elements of y appear in x?

#-# [1] TRUE

any(x > 0) #is any value in x larger than 0?

#-# [1] TRUE

all(x > 0) #are all values in x larger than 0?

#-# [1] TRUE

any(x > 2)

#-# [1] TRUE

all(x > 2)

#-# [1] FALSE

##############################
#Logical operators
#&& and || are called `double form' and & and | are called `single form' logical operators.

x==3 & y==3 #and
```

```
#-# [1] FALSE FALSE  TRUE FALSE FALSE

x==3 && y==3 #double form only evaluates first element of a vector

#-# [1] FALSE

x==3 | y==3 #or

#-# [1] TRUE TRUE TRUE TRUE TRUE

x==3 || y==3 #double form only evaluates first element of a vector

#-# [1] TRUE

# Missing values (NA) values are regarded as
# non-comparable even to themselves and will always return NA
# instead of TRUE or FALSE.
(y <- NA)

#-# [1] NA

y==5

#-# [1] NA

# So why would you use the double form? It seems useless.
# We would use the double form for reasons of efficiency.
# Evaluation proceeds only until the result is determined
# for the double form, whereas for the single form everything
# will be evaluated. Therefore the long form is more efficient.

#define fun1 to always return 5
fun1 <- function(){cat("fun1 executed \n"); return(5)}
#define fun2 to always return 6
fun2 <- function(){cat("fun2 executed  \n"); return(6)}

fun1()==5 | fun2()==6 # we see that both functions were executed

#-# fun1 executed
#-# fun2 executed
#-# [1] TRUE

fun1()==5 || fun2()==6 # we see that only fun1 is executed

#-# fun1 executed
#-# [1] TRUE

# Only fun1 is executed because only one of the two
# conditions needs to be TRUE and since the first
# condition is already TRUE we already know the answer.
```

2.4.3.2 Creating dummy variables

Many algorithms cannot handle categorical variables. Whenever we have a variable with nominal values we need to transform that variable in dummy, also called indicator, variables. Consider the dataset in Table 2.3. The result of the dummy transformation is displayed in Table 2.4.

Table 2.3: Categorical variables

CustomerID	gender	lifeCategory
5012953	m	youth
230329	f	mid
254823	f	mid
1943860	m	sen
82942	m	sen

Table 2.4: Dummy variables

CustomerID	female	male	lifeYouth	lifeMid	lifeSen
5012953	0	1	1	0	0
230329	1	0	0	1	0
254823	1	0	0	1	0
1943860	0	1	0	0	1
82942	0	1	0	0	1

The dataset in Table 2.4 is, however, not very efficient. We could remove one category of each original variable without losing any information. For example, gender is either male or female, so if we would delete the 'male' column, then we would still know if the customer is male or female by looking at the female column. If female equals 1 then the customer is a female, otherwise the customer is a male. The same goes for life category. We can remove one column without losing any information (e.g., 'lifeSen'). For predictive analytics it is unimportant as to which column to delete. The deleted categories will be represented by the intercept in a regression. If we remove the columns, 'lifeSen' and 'male' the intercept will represent male seniors. Table 2.5 represents an example of the final step in the creation of our dummy variables and represents the most efficient representation of our data. The columns that we remove are called reference variables.

Table 2.5: Dummy variables without reference variables

CustomerID	female	lifeYouth	lifeMid
5012953	0	1	0
230329	1	0	1
254823	1	0	1
1943860	0	0	0
82942	0	0	0

```r
#Create dummies manually
(x <- data.frame(ID=c(1,1,2,2,3,3),
                 V1=c(1,2,3,4,4,4)))

#-#   ID V1
#-# 1  1  1
#-# 2  1  2
#-# 3  2  3
#-# 4  2  4
#-# 5  3  4
#-# 6  3  4

x[,3] <- ifelse(x$V1==1,1,0)
x[,4] <- ifelse(x$V1==2,1,0)
x[,5] <- ifelse(x$V1==3,1,0)
x[,6] <- ifelse(x$V1==4,1,0)
x

#-#   ID V1 V3 V4 V5 V6
#-# 1  1  1  1  0  0  0
#-# 2  1  2  0  1  0  0
#-# 3  2  3  0  0  1  0
#-# 4  2  4  0  0  0  1
#-# 5  3  4  0  0  0  1
#-# 6  3  4  0  0  0  1

(names(x)[3:6] <- c('V1_1','V1_2','V1_3','V1_4'))

#-# [1] "V1_1" "V1_2" "V1_3" "V1_4"

#Create dummy variables automatically
if (!require('dummy')){
  install.packages('dummy',
                   repos="https://cran.rstudio.com/",
                   quiet=TRUE)
  require('dummy') #same as library(dummy)
}

#-# Loading required package: dummy
#-# dummy 0.1.3
#-# dummyNews()

(x <- data.frame(ID=c(1,1,1,1,2,2,2),
                 V1=as.character(c(1,2,3,4,4,4,4)),
                 V2=as.factor(c(1,2,3,4,4,4,4))))

#-#   ID V1 V2
#-# 1  1  1  1
#-# 2  1  2  2
#-# 3  1  3  3
#-# 4  1  4  4
#-# 5  2  4  4
#-# 6  2  4  4
#-# 7  2  4  4
```

```
(x <- dummy(x))

#-#   V1_1 V1_2 V1_3 V1_4 V2_1 V2_2 V2_3 V2_4
#-# 1    1    0    0    0    1    0    0    0
#-# 2    0    1    0    0    0    1    0    0
#-# 3    0    0    1    0    0    0    1    0
#-# 4    0    0    0    1    0    0    0    1
#-# 5    0    0    0    1    0    0    0    1
#-# 6    0    0    0    1    0    0    0    1
#-# 7    0    0    0    1    0    0    0    1

# Removing reference variables:
x$V2_4 <- x$V1_4 <- NULL

x

#-#   V1_1 V1_2 V1_3 V2_1 V2_2 V2_3
#-# 1    1    0    0    1    0    0
#-# 2    0    1    0    0    1    0
#-# 3    0    0    1    0    0    1
#-# 4    0    0    0    0    0    0
#-# 5    0    0    0    0    0    0
#-# 6    0    0    0    0    0    0
#-# 7    0    0    0    0    0    0
```

2.4.3.3 Conditional processing

```
#Consider the following self explanatory if then construct:
x <- 4

if (x <= 2){
  print("Smaller than or equal to 2")
} else if (x == 3) {
  print("Equal to 3")
} else {
  print("Greater than 3")
}

#-# [1] "Greater than 3"

#Note that && and || go with if()

#If x is a vector we need to use ifelse()
(x <- 1:5)

#-# [1] 1 2 3 4 5

ifelse(x == 4, 1, 0)

#-# [1] 0 0 0 1 0

#Note that & and | go with ifelse()
```

2.4.3.4 Timing code

If we want to make our code more efficient, we want to measure how long different parts take. There are two ways to do that.

```
#How long does one second take?
start <- Sys.time()
Sys.sleep("1")
round(Sys.time() - start)

#-# Time difference of 1 secs

#or
system.time(Sys.sleep("1"))

#-#    user  system elapsed
#-#   0.000   0.000   1.003
```

2.4.3.5 Loops

Instead of copy pasting code multiple times and changing it slightly, we use loops. This will make our code shorter and less error-prone. The 'for' is to be used when the number of iterations is known before executing the loop. Otherwise we can use the 'while or repeat loops'.

```
###### Loop types

#Consider three different loop constructs that print 10 numbers to the screen.

#For loop
for (i in 1:10){
  print(i)
}

#-# [1] 1
#-# [1] 2
#-# [1] 3
#-# [1] 4
#-# [1] 5
#-# [1] 6
#-# [1] 7
#-# [1] 8
#-# [1] 9
#-# [1] 10

#While loop
i <- 0
while (i < 10){
  i <- i + 1
  print(i)
}
```

```
#-# [1] 1
#-# [1] 2
#-# [1] 3
#-# [1] 4
#-# [1] 5
#-# [1] 6
#-# [1] 7
#-# [1] 8
#-# [1] 9
#-# [1] 10

#Repeat loop
i <- 0
repeat{
 i <- i + 1
 print(i)
 if (i==10) break
}

#-# [1] 1
#-# [1] 2
#-# [1] 3
#-# [1] 4
#-# [1] 5
#-# [1] 6
#-# [1] 7
#-# [1] 8
#-# [1] 9
#-# [1] 10

###### Storing values and applying functions to the
# columns of data.frames.

#Example 1:

# Loops are very slow. There is a way to make them faster:
# preallocation. If we do not preallocate R will,
# in each iteration: (1) copy the vector, and (2) append one
# value. Preallocation avoids this and is a lot faster.
# Before using a loop, we always need to ask ourselves if we
# really need it. If we are not putting a lot of code in
# our loop then probably there is a vectorized version of
# what we want to do. For example, consider the loop below.
# It simply stores the values 1 to 10 in an integer vector.
# Of course we know that we can do that by simply doing
# x <- 1:10. We have included the three alternatives just
# below:

#slow version
start <- Sys.time()
x <- integer()
```

```r
for (i in 1:100000){
  x[i] <- i
}
(duration1 <- Sys.time() - start)

#-# Time difference of 9.603038 secs

#faster version (use with preallocation)
start <- Sys.time()
x <- integer(100000)
for (i in 1:100000){
  x[i] <- i
}
(duration2 <- Sys.time() - start)

#-# Time difference of 0.07530093 secs

# Preallocation makes our code this many times faster:
as.numeric(duration1)/as.numeric(duration2)

#-# [1] 127.5288

#fastest version (use vectorized function)
start <- Sys.time()
x <- 1:100000
(duration3 <- Sys.time() - start)

#-# Time difference of 0.001312971 secs

# Using the vectorized solution in this case makes our code
# this many times faster:
as.numeric(duration2)/as.numeric(duration3)

#-# [1] 57.35155

#Example 2:

#create some data
x <- data.frame(matrix(runif(10000000),ncol=100000))
#this yields 100 rows and 100k columns
dim(x)

#-# [1]    100 100000

#Objective: compute the column sums

#Bad solution: Use a Loop to take the sum of
#each variable and save it (22 seconds on our system)
x_sum1 <- integer()
system.time(
for (i in 1:ncol(x)) x_sum1[i] <- sum(x[,i])
)
```

```
#-#    user  system elapsed
#-#  15.574  10.521  26.114

head(x_sum1)

#-# [1] 52.27687 51.59539 49.90541 49.99103 47.10923 51.70440

#Better solution: Pre-allocate the object and
#use a Loop to take the sum of each variable and
#save it (6 seconds on our system)
x_sum2 <- integer(length=100000)
system.time(
  for (i in 1:ncol(x)) x_sum2[i] <- sum(x[,i])
)

#-#    user  system elapsed
#-#   6.758   0.009   6.772

head(x_sum2)

#-# [1] 52.27687 51.59539 49.90541 49.99103 47.10923 51.70440

#Best solution: Use sapply (0.1 seconds on our system)
#Conceptually sapply successively takes the columns of
#the data frame, applies a function to each column,
#returns a vector of the same length as the number of
#columns.
system.time(
x_sum3 <- sapply(x,sum)
)

#-#    user  system elapsed
#-#   0.094   0.000   0.095

head(x_sum3)

#-#       X1       X2       X3       X4       X5       X6
#-# 52.27687 51.59539 49.90541 49.99103 47.10923 51.70440

#There are some ready made function that rely on sapply
head(colSums(x))

#-#       X1       X2       X3       X4       X5       X6
#-# 52.27687 51.59539 49.90541 49.99103 47.10923 51.70440

head(colMeans(x))

#-#        X1        X2        X3        X4        X5        X6
#-# 0.5227687 0.5159539 0.4990541 0.4999103 0.4710923 0.5170440
```

```r
###### Error handling

# Consider the following values
vals <- list(1, 8, 4, -1, 'one', 0, 10)

#We are going to loop through it.
#In each iteration we compute the logarithm

for(i in vals) {
  cat("log(", i, ")=", log(i),"\n")
}

#-# log( 1 )= 0
#-# log( 8 )= 2.079442
#-# log( 4 )= 1.386294

#-# Warning in log(i): NaNs produced

#-# log( -1 )= NaN

#-# Error in log(i): non-numeric argument to mathematical function

# The loop exits early without going through all
# the values. It produces a warning when
# i=-1 and an error when i='one'.

# If we would have been doing some
# heavy computations in our loop and we
# would have it run overnight, we would find
# an error message in the morning.
# It would be much better if R
# would simply skip the error and move on to
# the next iteration. R would do as much iterations
# as possible and we can fix the errors afterwards.
# Essentially we are telling R to keep going and skip
# iterations that produce an error so we can
# investigate the errors later, as opposed to
# when they are produced.

# Skipping errors can be accomplished with the try
# function as follows:

for(i in vals) {
  try(cat("log(", i, ")=", log(i),"\n"))
}

#-# log( 1 )= 0
#-# log( 8 )= 2.079442
#-# log( 4 )= 1.386294

#-# Warning in log(i): NaNs produced
```

```
#-# log( -1 )= NaN
#-# log( 0 )= -Inf
#-# log( 10 )= 2.302585

# In some cases we want some more power. We might
# want to write our own error or warning handlers.
# For example, we may want to add 1 to i when i=-1.
# If we want to do that we can use the tryCatch
# function.

for (i in vals) {
  tryCatch(
    cat("log(", i, ")=", log(i),"\n"),
    error=function(x){
        cat("log(", i, ")=ERROR :",conditionMessage(x), "\n")},
    warning=function(x){
        cat("log(", i, ")=WARNING :",conditionMessage(x),
            "/// Compute log(", i ,"+1) instead=",
            log(i+1),"\n")})
}

#-# log( 1 )= 0
#-# log( 8 )= 2.079442
#-# log( 4 )= 1.386294
#-# log( -1 )=WARNING : NaNs produced /// Compute log( -1 +1) instead= -Inf
#-# log( one )=ERROR : non-numeric argument to mathematical function
#-# log( 0 )= -Inf
#-# log( 10 )= 2.302585
```

2.4.3.6 Applying functions to lists

```
# We could use a loop, but we can also use the following two functions:
# -lapply
# -do.call

#Consider the following list
df1 <- data.frame(a=c(1,2),b=c(3,4),c=c(5,6))
df2 <- data.frame(a=c(7,8),b=c(9,10),c=c(11,12))

(l <- list(df1,df2))

#-# [[1]]
#-#   a b c
#-# 1 1 3 5
#-# 2 2 4 6
#-#
#-# [[2]]
#-#   a  b  c
```

```
#-# 1 7  9 11
#-# 2 8 10 12

# Suppose that we want to stack df1 and df2
# We would rather use do.call and not use lapply
# Example:

# lapply applies a function over a list
lapply(l,rbind)

#-# [[1]]
#-#   a b c
#-# 1 1 3 5
#-# 2 2 4 6
#-#
#-# [[2]]
#-#   a  b  c
#-# 1 7  9 11
#-# 2 8 10 12

# That is equivalent to
list(rbind(l[[1]]),rbind(l[[2]]))

#-# [[1]]
#-#   a b c
#-# 1 1 3 5
#-# 2 2 4 6
#-#
#-# [[2]]
#-#   a  b  c
#-# 1 7  9 11
#-# 2 8 10 12

# do.call calls a function with a list of arguments
do.call(rbind,l)

#-#   a  b  c
#-# 1 1  3  5
#-# 2 2  4  6
#-# 3 7  9 11
#-# 4 8 10 12

# The latter is equivalent to
rbind(l[[1]],l[[2]])

#-#   a  b  c
#-# 1 1  3  5
#-# 2 2  4  6
#-# 3 7  9 11
#-# 4 8 10 12
```

```
# Hence it makes no sence to combine rbind and lapply.
# However, lapply could be useful for other things, such as:
lapply(l,colMeans)

#-# [[1]]
#-#   a   b   c
#-# 1.5 3.5 5.5
#-#
#-# [[2]]
#-#    a    b    c
#-#  7.5  9.5 11.5
```

2.4.3.7 Aggregating

```
#Create a data frame
(x <- data.frame(ID=c(1,1,1,1,2,2,2),
                 V1=c(1,2,3,4,4,4,4),
                 V2=c(1,2,3,4,4,4,4)))

#-#   ID V1 V2
#-# 1  1  1  1
#-# 2  1  2  2
#-# 3  1  3  3
#-# 4  1  4  4
#-# 5  2  4  4
#-# 6  2  4  4
#-# 7  2  4  4

?aggregate

#sum
aggregate(x[,names(x) != 'ID'],by=list(ID=x$ID),sum)

#-#   ID V1 V2
#-# 1  1 10 10
#-# 2  2 12 12

#counts
aggregate(x[,names(x) != 'ID'],by=list(x$ID),length)

#-#   Group.1 V1 V2
#-# 1       1  4  4
#-# 2       2  3  3

#Notice the difference in the by argument
#between the first and last aggregate

#first or last observation
aggregate(x[,names(x) != 'ID'],by=list(x$ID),head,n=1)
```

```
#-#   Group.1 V1 V2
#-# 1       1  1  1
#-# 2       2  4  4

aggregate(x[,names(x) != 'ID'],by=list(x$ID),tail,n=1)

#-#   Group.1 V1 V2
#-# 1       1  4  4
#-# 2       2  4  4
```

2.4.3.8 Merging

```
#Merging 2 data frames
# ?merge

#Create two data frames
(x <- data.frame(ID=c(1,1,2,2,3,3),
                 V1=c(1,2,3,4,4,4),
                 V2=c(1,2,3,4,4,4)))

#-#   ID V1 V2
#-# 1  1  1  1
#-# 2  1  2  2
#-# 3  2  3  3
#-# 4  2  4  4
#-# 5  3  4  4
#-# 6  3  4  4

(y <- data.frame(ID=c(2,2,3,3,4,4),
                 V3=c(1,2,3,4,4,4),
                 V4=c(1,2,3,4,4,4)))

#-#   ID V3 V4
#-# 1  2  1  1
#-# 2  2  2  2
#-# 3  3  3  3
#-# 4  3  4  4
#-# 5  4  4  4
#-# 6  4  4  4

(x_by <- aggregate(x[,names(x) != 'ID'],
                   by=list(ID=x$ID),sum))

#-#   ID V1 V2
#-# 1  1  3  3
#-# 2  2  7  7
#-# 3  3  8  8
```

```
(y_by <- aggregate(y[,names(y) != 'ID'],
                   by=list(y$ID),sum))

#-#   Group.1 V3 V4
#-# 1       2  3  3
#-# 2       3  7  7
#-# 3       4  8  8

#You can name a column like this:
names(y_by)[1] <- 'ID'
#or you can list(ID=x$ID) as for x_by

str(x_by)

#-# 'data.frame': 3 obs. of  3 variables:
#-#  $ ID: num  1 2 3
#-#  $ V1: num  3 7 8
#-#  $ V2: num  3 7 8

str(y_by)

#-# 'data.frame': 3 obs. of  3 variables:
#-#  $ ID: num  2 3 4
#-#  $ V3: num  3 7 8
#-#  $ V4: num  3 7 8

#Inner Join
merge(x_by,y_by,by='ID')

#-#   ID V1 V2 V3 V4
#-# 1  2  7  7  3  3
#-# 2  3  8  8  7  7

#Left outer join
merge(x_by,y_by,by='ID',all.x=TRUE)

#-#   ID V1 V2 V3 V4
#-# 1  1  3  3 NA NA
#-# 2  2  7  7  3  3
#-# 3  3  8  8  7  7

#Right outer join
merge(x_by,y_by,by='ID',all.y=TRUE)

#-#   ID V1 V2 V3 V4
#-# 1  2  7  7  3  3
#-# 2  3  8  8  7  7
#-# 3  4 NA NA  8  8

#Full outer join
merge(x_by,y_by,by='ID',all=TRUE)
```

```
#-#   ID V1 V2 V3 V4
#-# 1  1  3  3 NA NA
#-# 2  2  7  7  3  3
#-# 3  3  8  8  7  7
#-# 4  4 NA NA  8  8

#Merging multiple data frames

#create a third dataset
(z <- data.frame(ID=c(2,2,3,3,4,4),
                 V5=c(1,2,3,4,4,4),
                 V6=c(1,2,3,4,4,4)))

#-#   ID V5 V6
#-# 1  2  1  1
#-# 2  2  2  2
#-# 3  3  3  3
#-# 4  3  4  4
#-# 5  4  4  4
#-# 6  4  4  4

(z_by <- aggregate(z[,names(z) != 'ID'],
                   by=list(ID=z$ID),sum))

#-#   ID V5 V6
#-# 1  2  3  3
#-# 2  3  7  7
#-# 3  4  8  8

#Approach 1: Use a loop (the bad approach, correct but slower)
datalist <- list(x_by,y_by,z_by)
for (i in 1:length(datalist)) {
  if (i==1) intermediate <- datalist[[1]]
  if (i>1) intermediate <- merge(intermediate,
                                 datalist[[i]],by='ID')
}
intermediate

#-#   ID V1 V2 V3 V4 V5 V6
#-# 1  2  7  7  3  3  3  3
#-# 2  3  8  8  7  7  7  7

#Approach 2: Reduce (the good approach)
#?Reduce

#The Reduce function successively combines the elements
#of a given vector. To see the process in action,
#set accumulate=TRUE
str(datalist)
```

```
#-# List of 3
#-#  $ :'data.frame': 3 obs. of  3 variables:
#-#   ..$ ID: num [1:3] 1 2 3
#-#   ..$ V1: num [1:3] 3 7 8
#-#   ..$ V2: num [1:3] 3 7 8
#-#  $ :'data.frame': 3 obs. of  3 variables:
#-#   ..$ ID: num [1:3] 2 3 4
#-#   ..$ V3: num [1:3] 3 7 8
#-#   ..$ V4: num [1:3] 3 7 8
#-#  $ :'data.frame': 3 obs. of  3 variables:
#-#   ..$ ID: num [1:3] 2 3 4
#-#   ..$ V5: num [1:3] 3 7 8
#-#   ..$ V6: num [1:3] 3 7 8

Reduce(function(x, y) merge(x, y, by='ID'),datalist,accumulate=TRUE)

#-# [[1]]
#-#   ID V1 V2
#-# 1  1  3  3
#-# 2  2  7  7
#-# 3  3  8  8
#-#
#-# [[2]]
#-#   ID V1 V2 V3 V4
#-# 1  2  7  7  3  3
#-# 2  3  8  8  7  7
#-#
#-# [[3]]
#-#   ID V1 V2 V3 V4 V5 V6
#-# 1  2  7  7  3  3  3  3
#-# 2  3  8  8  7  7  7  7

#To keep only the final merge drop the
#accumulate (default is accumulate=FALSE)
Reduce(function(x, y) merge(x, y, by='ID'),datalist)

#-#   ID V1 V2 V3 V4 V5 V6
#-# 1  2  7  7  3  3  3  3
#-# 2  3  8  8  7  7  7  7
```

2.4.3.9 Working with dates

```
#When we read in data, dates are read in as a
#character vector. The function as.Date() creates
#a Date object. Most systems store dates internally
#as the number of days (numeric vector) since some
#origin. When using as.Date on a numeric vector we
#need to set the origin. The origin on this system
```

```
#is "1970/01/01"
help(as.Date)

#This is how you find out what the origin is:
Sys.Date()-as.integer(Sys.Date())

#-# [1] "1970-01-01"

#This is Today's date:
Sys.Date()

#-# [1] "2017-04-06"

#Converting an integer requires an origin:
as.Date(10,origin="1970/01/01")

#-# [1] "1970-01-11"

# formats
# %d  day as a number
# %a  abbreviated weekday
# %A  unabbreviated weekday
# %m month
# %b  abbreviated month
# %B unabbreviated month
# %y  2-digit year
# %Y 4-digit year

#More information:
# help(strftime)
# help(ISOdatetime)

#Specifying a date format:
f <- "%d/%m/%Y"

#Subtraction of two dates as Date object
#using the format:
as.Date('02/01/1970',format=f) -
  as.Date('01/01/1970',format=f)

#-# Time difference of 1 days

#Subtraction of two dates as numeric objects:
as.numeric(as.Date('02/01/1970',format=f)) -
  as.numeric(as.Date('01/01/1970',format=f))

#-# [1] 1

#Adding zero days to the origin should be 0:
as.numeric(as.Date(0,origin="1970/01/01"))
```

```
#-# [1] 0

#Read in the data with dates as characters
URL_subs <- "http://ballings.co/hidden/aCRM/data/chapter2/subscriptions.txt"

subscriptions <-
  read.table(URL_subs,
             header=TRUE, sep=";",
             colClasses=c("character",
                          "character",
                          "character",
                          "character",
                          "character",
                          "character",
                          "integer",
                          "integer",
                          "character",
                          "factor",
                          "factor",
                          "character",
                          "character",
                          "numeric",
                          "numeric",
                          "numeric",
                          "numeric",
                          "numeric",
                          "numeric",
                          "numeric",
                          "numeric"))

#Transform the dates in the subscriptions data frame
#First look at the data
str(subscriptions,vec.len=0.7)

#-# 'data.frame': 227 obs. of  21 variables:
#-#  $ SubscriptionID   : chr  "1000024" ...
#-#  $ CustomerID       : chr  "1150045" ...
#-#  $ ProductID        : chr  "8" ...
#-#  $ Pattern          : chr  "1111110" ...
#-#  $ StartDate        : chr  "18/01/2010" ...
#-#  $ EndDate          : chr  "17/04/2010" ...
#-#  $ NbrNewspapers    : int  76 50 ...
#-#  $ NbrStart         : int  25 25 ...
#-#  $ RenewalDate      : chr  "17/03/2010" ...
#-#  $ PaymentType      : Factor w/ 2 levels "BT","DD": 1 1 ...
#-#  $ PaymentStatus    : Factor w/ 2 levels "Not Paid","Paid": 2 2 ...
#-#  $ PaymentDate      : chr  "27/01/2010" ...
#-#  $ FormulaID        : chr  "8552" ...
#-#  $ GrossFormulaPrice: num  79 ...
#-#  $ NetFormulaPrice  : num  79 29.6 ...
#-#  $ NetNewspaperPrice: num  1.04 ...
```

```
#-#  $ ProductDiscount  : num  0 0 ...
#-#  $ FormulaDiscount  : num  0 ...
#-#  $ TotalDiscount    : num  0 ...
#-#  $ TotalPrice       : num  79 29.6 ...
#-#  $ TotalCredit      : num  0 0 ...

head(subscriptions$StartDate)

#-# [1] "18/01/2010" "14/01/2010" "01/01/2008" "02/01/2007"
#-# [5] "01/01/2008" "02/01/2007"

#Get the positions of the date variables
(vars <- which(names(subscriptions) %in%
               c("StartDate","EndDate","PaymentDate","RenewalDate")))

#-# [1]  5  6  9 12

#Transform to dates
subscriptions[,vars] <- sapply(vars,function(vars) {
                                as.Date(subscriptions[,vars],format="%d/%m/%Y")},
                           simplify=FALSE)
#Note: in this case 'simplify' means simplify the result
#to a numeric vector, which is just as good here
str(subscriptions,vec.len=0.7)

#-# 'data.frame': 227 obs. of  21 variables:
#-#  $ SubscriptionID   : chr  "1000024" ...
#-#  $ CustomerID       : chr  "1150045" ...
#-#  $ ProductID        : chr  "8" ...
#-#  $ Pattern          : chr  "1111110" ...
#-#  $ StartDate        : Date, format: "2010-01-18" ...
#-#  $ EndDate          : Date, format: "2010-04-17" ...
#-#  $ NbrNewspapers    : int  76 50 ...
#-#  $ NbrStart         : int  25 25 ...
#-#  $ RenewalDate      : Date, format: "2010-03-17" ...
#-#  $ PaymentType      : Factor w/ 2 levels "BT","DD": 1 1 ...
#-#  $ PaymentStatus    : Factor w/ 2 levels "Not Paid","Paid": 2 2 ...
#-#  $ PaymentDate      : Date, format: "2010-01-27" ...
#-#  $ FormulaID        : chr  "8552" ...
#-#  $ GrossFormulaPrice: num  79 ...
#-#  $ NetFormulaPrice  : num  79 29.6 ...
#-#  $ NetNewspaperPrice: num  1.04 ...
#-#  $ ProductDiscount  : num  0 0 ...
#-#  $ FormulaDiscount  : num  0 ...
#-#  $ TotalDiscount    : num  0 ...
#-#  $ TotalPrice       : num  79 29.6 ...
#-#  $ TotalCredit      : num  0 0 ...

head(subscriptions$StartDate)

#-# [1] "2010-01-18" "2010-01-14" "2008-01-01" "2007-01-02"
#-# [5] "2008-01-01" "2007-01-02"
```

```
#Reading in dates directly (as opposed to changing them afterwards)
#what is the current default date format?
Sys.Date()

#-# [1] "2017-04-06"

#looks like it is YYYY-MM-DD

#Currently all the dates are character vectors
#The date format in our data is DD/MM/YYYY (e.g.,"27/01/2010")


#If we use Date, values are not imported adequately.
#Let's try it. Look at StartDate
subscriptions <- read.table(URL_subs,
                            header=TRUE, sep=";",
                            colClasses=c("integer",
                                         "integer",
                                         "character",
                                         "character",
                                         "Date",
                                         "character",
                                         "integer",
                                         "integer",
                                         "character",
                                         "factor",
                                         "factor",
                                         "character",
                                         "integer",
                                         "numeric",
                                         "numeric",
                                         "numeric",
                                         "numeric",
                                         "numeric",
                                         "numeric",
                                         "numeric",
                                         "numeric"))
str(subscriptions,vec.len=0.7)

#-# 'data.frame': 227 obs. of  21 variables:
#-#  $ SubscriptionID   : int  1000024 1000082 ...
#-#  $ CustomerID       : int  1150045 1150046 ...
#-#  $ ProductID        : chr  "8" ...
#-#  $ Pattern          : chr  "1111110" ...
#-#  $ StartDate        : Date, format: "0018-01-20" ...
#-#  $ EndDate          : chr  "17/04/2010" ...
#-#  $ NbrNewspapers    : int  76 50 ...
#-#  $ NbrStart         : int  25 25 ...
#-#  $ RenewalDate      : chr  "17/03/2010" ...
#-#  $ PaymentType      : Factor w/ 2 levels "BT","DD": 1 1 ...
#-#  $ PaymentStatus    : Factor w/ 2 levels "Not Paid","Paid": 2 2 ...
```

```
#-#  $ PaymentDate      : chr  "27/01/2010" ...
#-#  $ FormulaID        : int  8552 10017 ...
#-#  $ GrossFormulaPrice: num  79 ...
#-#  $ NetFormulaPrice  : num  79 29.6 ...
#-#  $ NetNewspaperPrice: num  1.04 ...
#-#  $ ProductDiscount  : num  0 0 ...
#-#  $ FormulaDiscount  : num  0 ...
#-#  $ TotalDiscount    : num  0 ...
#-#  $ TotalPrice       : num  79 29.6 ...
#-#  $ TotalCredit      : num  0 0 ...

head(subscriptions$StartDate)

#-# [1] "0018-01-20" "0014-01-20" "0001-01-20" "0002-01-20"
#-# [5] "0001-01-20" "0002-01-20"

#StartDate clearly read incorrectly

#Take care of default date format
f <- "%d/%m/%Y"
setClass('fDate')
setAs(from="character",
      to="fDate",
      def=function(from) as.Date(from, format=f) )

#Let's use fDate instead of Date for StartDate
#Look at StartDate
subscriptions <- read.table(URL_subs,
                            header=TRUE, sep=";",
                            colClasses=c("integer",
                                         "integer",
                                         "character",
                                         "character",
                                         "fDate",
                                         "character",
                                         "integer",
                                         "integer",
                                         "character",
                                         "factor",
                                         "factor",
                                         "character",
                                         "integer",
                                         "numeric",
                                         "numeric",
                                         "numeric",
                                         "numeric",
                                         "numeric",
                                         "numeric",
                                         "numeric",
                                         "numeric"))
str(subscriptions,vec.len=0.7)
```

```
#-# 'data.frame': 227 obs. of  21 variables:
#-#  $ SubscriptionID   : int  1000024 1000082 ...
#-#  $ CustomerID       : int  1150045 1150046 ...
#-#  $ ProductID        : chr  "8" ...
#-#  $ Pattern          : chr  "1111110" ...
#-#  $ StartDate        : Date, format: "2010-01-18" ...
#-#  $ EndDate          : chr  "17/04/2010" ...
#-#  $ NbrNewspapers    : int  76 50 ...
#-#  $ NbrStart         : int  25 25 ...
#-#  $ RenewalDate      : chr  "17/03/2010" ...
#-#  $ PaymentType      : Factor w/ 2 levels "BT","DD": 1 1 ...
#-#  $ PaymentStatus    : Factor w/ 2 levels "Not Paid","Paid": 2 2 ...
#-#  $ PaymentDate      : chr  "27/01/2010" ...
#-#  $ FormulaID        : int  8552 10017 ...
#-#  $ GrossFormulaPrice: num  79 ...
#-#  $ NetFormulaPrice  : num  79 29.6 ...
#-#  $ NetNewspaperPrice: num  1.04 ...
#-#  $ ProductDiscount  : num  0 0 ...
#-#  $ FormulaDiscount  : num  0 ...
#-#  $ TotalDiscount    : num  0 ...
#-#  $ TotalPrice       : num  79 29.6 ...
#-#  $ TotalCredit      : num  0 0 ...

head(subscriptions$StartDate)

#-# [1] "2010-01-18" "2010-01-14" "2008-01-01" "2007-01-02"
#-# [5] "2008-01-01" "2007-01-02"

#StartDate is correct now
```

In sum, dates are read in as characters. We can transform them afterwards, but it's more efficient to read them in correctly from the first time. We can use the `Date` class if the dates are stored in the default format. If they are not we need to define our own class and use it when reading in data.

2.4.3.10 Creating functions

Whenever we want to reuse a piece of code multiple times in the future we make a function. This will ensure minimal code maintenance.

```
#Make a new function:

# functionname <- fuction(parametername){
#   code
# }

#Example:
MySum <- function(a,b){
  c <- a+b
```

```
  c #since this is the last line, c will be returned
}

MySum(1,2)

#-# [1] 3

#This is identical:
MySum <- function(a,b){
  c <- a+b
  return(c)
}

MySum(1,2)

#-# [1] 3

#Everything we do in a function stays in
#the function, unless we make it global.

outerfunc <- function(a,b){

  innerfunc <- function(a,b){
    a+b
  }
  v <- 1
}

outerfunc(1,2) #Nothing is printed (it does return v but invisibly)

(d <- outerfunc(1,2))

#-# [1] 1

#What comes next will not work.
#R will complain that it
#cannot find the function and object
innerfunc(1,2)

#-# Error in eval(expr, envir, enclos): could not find function "innerfunc"

v

#-# Error in eval(expr, envir, enclos): object 'v' not found

#Make the innerfunc and v global
outerfunc <- function(a,b){

  innerfunc <<- function(a,b){
    a+b
  }
  v <<- 1
}

outerfunc(1,2)
innerfunc(1,2)
```

```
#-# [1] 3

v

#-# [1] 1

#However, this is generally considered bad practice.
#It's better to have a function return another function
outerfunc <- function(){

  innerfunc <- function(a,b){
    a+b
  }
  v <- 1
  return(list(innerfunc,v))
}

f <- outerfunc()
f

#-# [[1]]
#-# function (a, b)
#-# {
#-#     a + b
#-# }
#-# <environment: 0x7fc70ab99078>
#-#
#-# [[2]]
#-# [1] 1

f[[1]](1,2)

#-# [1] 3

f[[2]]

#-# [1] 1
```

2.4.3.11 Text Mining

Until now we have only considered structured data. We have learned how to explore those data and prepare them for the modeling phase. If we have unstructured data such as text we need to do more work. Figure 2.7 provides an overview of the different steps involved in text mining. Documents can be emails, tweets, posts, letters; or any other textual communication. 'Terms' refers to words in these documents. 'Rank' refers to the number of columns. We will discuss LSI and CV later in this section. In what follows we focus on the text pre-processing and dimension reduction steps.

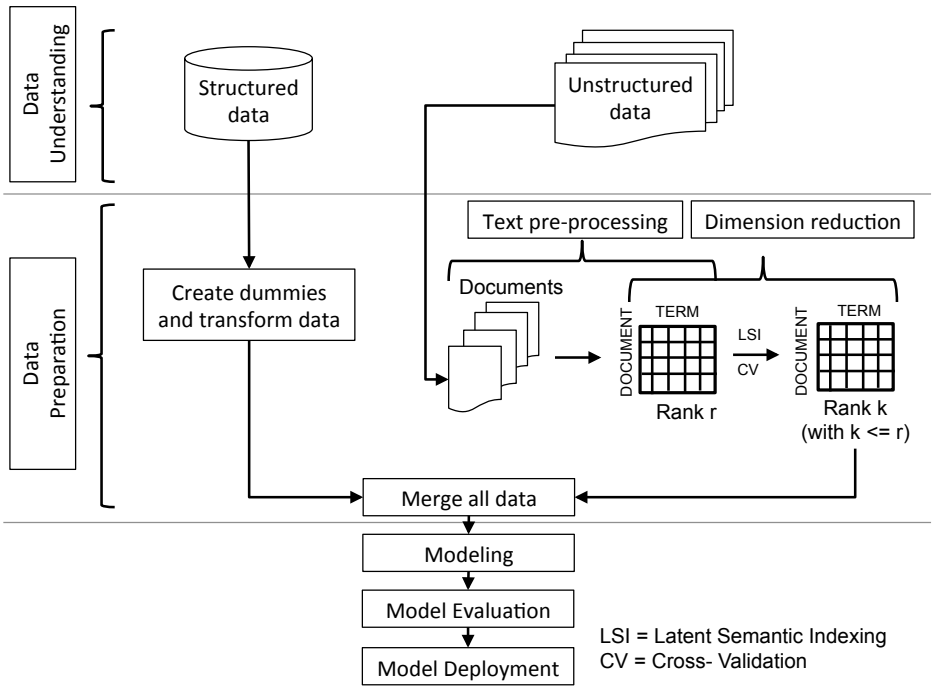


Figure 2.7: Text mining in perspective (adapted from Coussement and Van den Poel, 2008)

Text pre-processing

Consider the three documents in Table 2.6. They are suggestions from customers (denoted by ID) submitted through a bank's feedback system. The first step is to list all the unique words in all documents: online, banking, could, be, more, user-friendly, services, faster, alerts. The second step it to create a matrix (Table 2.7), called the document-by- term matrix. The cells in the matrix denote the number of occurences of a given word (see Table 2.7).

Table 2.6: Example of documents

ID	Suggestion
1	online banking could be more user-friendly
2	online banking services could be faster
3	faster alerts, faster services

Table 2.7: Example document-by-term matrix

ID	online	banking	could	be	more	user-friendly	services	faster	alerts
1	1	1	1	1	1	1	0	0	0
2	1	1	1	1	0	0	1	1	0
3	0	0	0	0	0	0	1	2	1

What is special about this matrix is that it is sparse. A total of 12 out of 27 cells have zero: 44% is empty. In addition $p > n$, with p the number of columns and n the number of instances. We can reduce the sparsity of the document-by-term matrix (i.e., increase its information density) by reducing the number of columns by further processing the text. It is important to note that we will alter the text documents, as opposed to the document-by-term matrix. In the next subsection 'Dimension reduction' we will see how we can further reduce the number of columns by making linear combinations of the columns. Here, we will alter the document-by-term matrix and not the text documents.

There are five steps involved in text pre-processing. There can be additional steps (e.g., synonym replacement), but these are out of the scope of this book.

1. Raw text cleaning: case conversion (transform everything to lower case), remove punctuation and stopwords, and do a spelling check
2. Stemming: replace items by their stem (e.g., connected, connecting, connection, ..., become connect)
3. Create document-by-term matrix
4. Term filtering: remove low frequency words; word frequencies follow a Zipfian (also called Zipf) distribution (half of the words appear only once or twice)
5. Document vector weighting

Document vector weighting is explained in more detail in the following code. The following code is available from `http://ballings.co/hidden/aCRM/code/chapter2/TextMining.R`

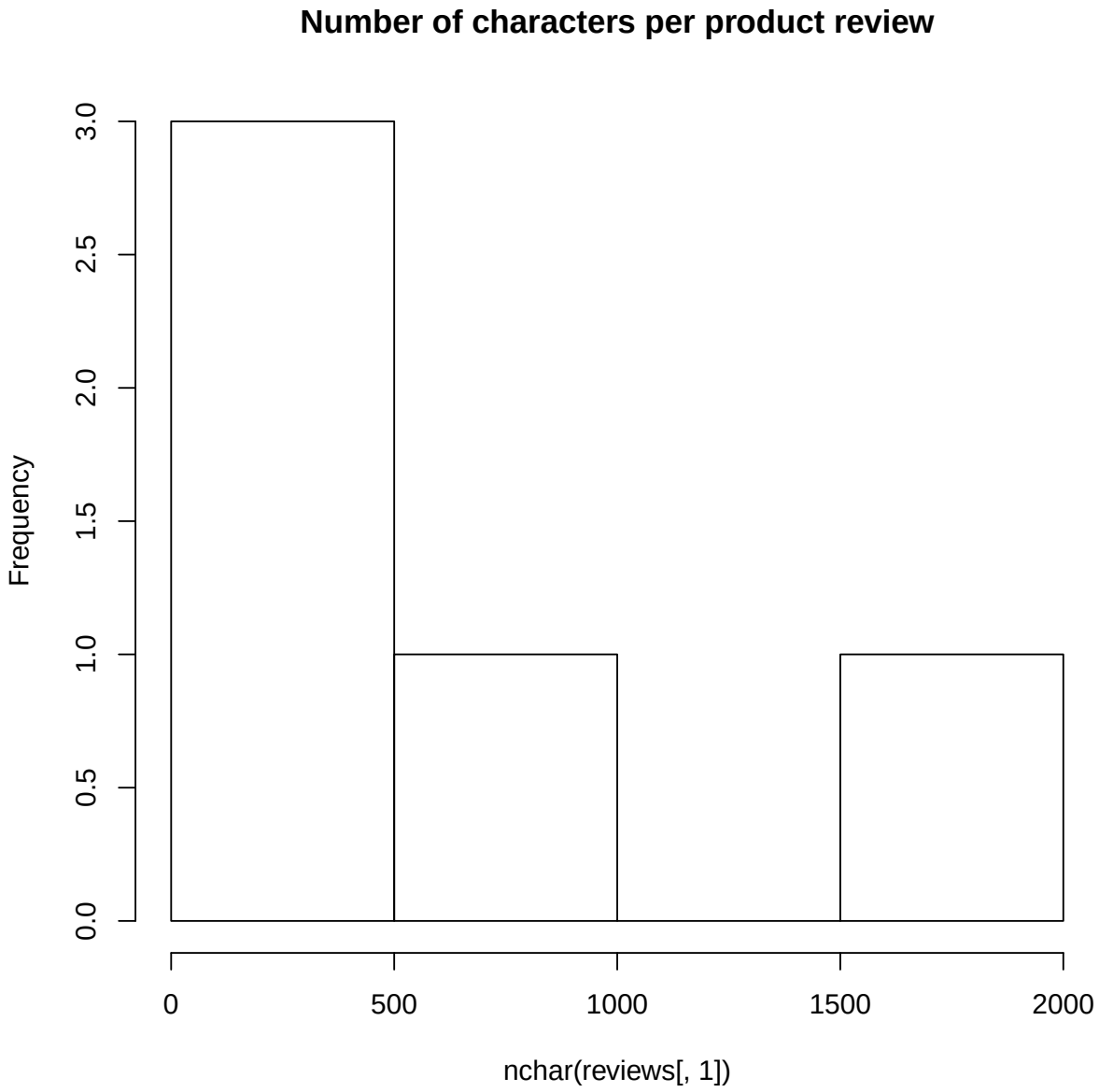
```
# How to give rare terms more weight?

#consider the following document (in the rows) by term matrix (in the columns)
dtm <- matrix(c(0,1,3,0,2,2,2,0,2,0,0,0,6,0,6,0,8,7,8,7),
              ncol=5)
colnames(dtm) <- c("term1","term2","term3","term4","term5")
rownames(dtm) <- c("doc1","doc2","doc3","doc4")
dtm

#-#      term1 term2 term3 term4 term5
#-# doc1     0     2     2     6     8
#-# doc2     1     2     0     0     7
#-# doc3     3     2     0     6     8
#-# doc4     0     0     0     0     7

# The values in the dtm are called weights and are at this point simply the raw
# frequencies of appearance in a document.

# w_ij= tf_ij
# The weight of a term i in document j equals the term
# frequency of term i in document j
```

```
# If we want to give more weight to rare terms we can multiply tf_ij
# with the inverse document frequency
# w_ij= tf_ij * idf_i
# with
# tf_ij the term frequencies of term i in document j:
# idf_i the inverse document frequencies of term i
# and
# idf_i= n/df_i
# with n equal to the total number of documents
# and df equal to the number of documents where term i was present

# Linking back to our example

#We notice that the
#-first term appears in 2 documents
#-second term appears in 3 documents
#-third term appears in 1 document
#-fourth term appears in 2 documents
#-fifth term appreas in 4 documents

#So we would expect that the third term would get the highest idf and that
# the fifth term would get the lowest idf
(idf <- nrow(dtm)/colSums(dtm >= 1))

#-#    term1    term2    term3    term4    term5
#-# 2.000000 1.333333 4.000000 2.000000 1.000000

#and that is exactly what we get

#transform vector of idf to a matrix so we can multiply it elementwise
(idf_mat <- matrix(idf[sapply(1:ncol(dtm),
                  function(x) rep(x,nrow(dtm)))],nrow=nrow(dtm)))

#-#      [,1]     [,2] [,3] [,4] [,5]
#-# [1,]    2 1.333333    4    2    1
#-# [2,]    2 1.333333    4    2    1
#-# [3,]    2 1.333333    4    2    1
#-# [4,]    2 1.333333    4    2    1

(dtm_weighted <- dtm * idf_mat)

#-#      term1    term2 term3 term4 term5
#-# doc1     0 2.666667     8    12     8
#-# doc2     2 2.666667     0     0     7
#-# doc3     6 2.666667     0    12     8
#-# doc4     0 0.000000     0     0     7

#In dtm_weighted the weight of a term is inversely related to
#the number of documents in which the term occured.

#Final remark: to reduce the impact of the length of different
```

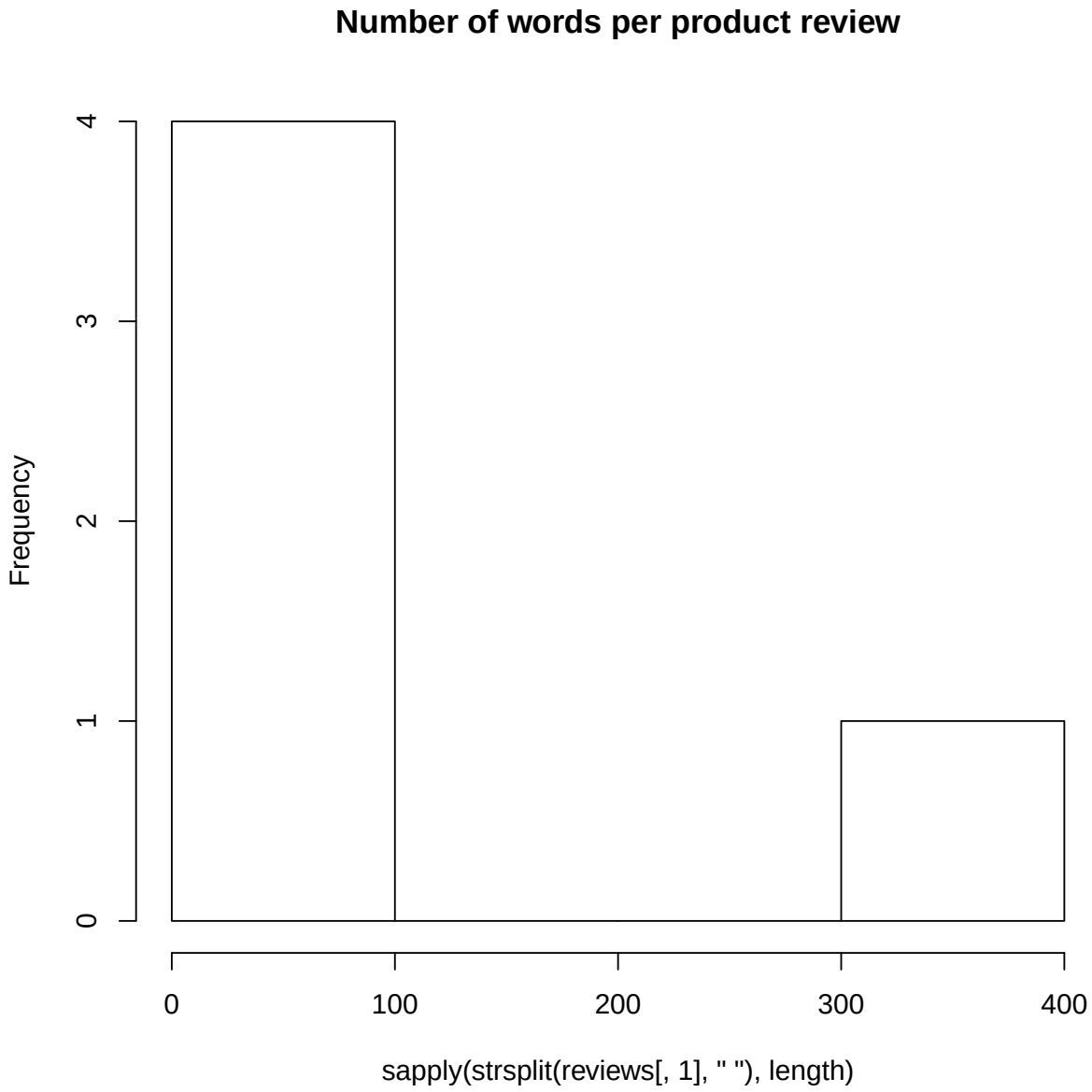
```
#documents we can apply the logarithm to the tf values.
#tf_ij= log1p(tf_ij)

#We can also reduce the effect of the raw idf by taking the logarithm
# idf_i= log(n/df_i) + 1
#We add a 1 for the cases were the term appears in all doucments (n=df_i).
# The idf will be 1 and the log(1)=0.

#This is the final weighted dtm:
(dtm_weighted <- log1p(dtm) * (log(idf_mat)+1))

#-#       term1    term2    term3    term4    term5
#-# doc1 0.0000 1.414663 2.621612 3.294712 2.197225
#-# doc2 1.1736 1.414663 0.000000 0.000000 2.079442
#-# doc3 2.3472 1.414663 0.000000 3.294712 2.197225
#-# doc4 0.0000 0.000000 0.000000 0.000000 2.079442
```

In what follows we provide an overview of how to use *R* to mine text.

```
#First read in data about product reviews by customers.
URL_prod_rev <-
  "http://www.ballings.co/hidden/aCRM/data/chapter2/productreviews.csv"
reviews <- read.csv(URL_prod_rev,
                    header=FALSE,
                    sep="\n",
                    stringsAsFactors=FALSE)
#When using sep="\n" it is assumed that one wants
#to read in entire lines verbatim.

#Take a subset of these reviews for demonstration purposes
reviews <- reviews[1:5,,drop=FALSE]

#How many reviews are there
nrow(reviews)

#-# [1] 5

#Let's look at the first few reviews.
#Only display the first 50 characters.
sapply(head(reviews),function(x) substr(x,1,50))

#-#      V1
#-# [1,] "Two month-long trips abroad: this is the best. It "
#-# [2,] "This is nearly as heavy as my laptop and I was hop"
#-# [3,] "Wonderfully thin, light, and durable. The keyboard"
#-# [4,] "This Keyboard/case cover is Absolutely FABULOUS!!!"
#-# [5,] "Great case! Easy to use, thin, and turns my iPad i"

#How many characters does each product review contain?
hist(nchar(reviews[,1]),
     main="Number of characters per product review")
```




```
#Most of the reviews contains 500 characters or less

#How many words does each product review contain?
hist(sapply(strsplit(reviews[,1]," "),length),
    main="Number of words per product review")
```




```
#Most of the reviews contains 100 words or less

#Since we have only one column we
#store it as a character vector:
reviews <- reviews[,1]

#Now that we have a good understanding of the data
#we can move on to data preparation

# Install the tm (i.e., text mining) package.
# The tm package expects the SnowballC and slam packages to be installed.
```

```
for (i in c('SnowballC','slam','tm')){
  if (!require(i, character.only=TRUE)) install.packages(i, repos = "http://cran.us.r-project.org")
  require(i, character.only=TRUE)
}

#-# Loading required package: SnowballC
#-# Loading required package: slam
#-# Loading required package: tm
#-# Loading required package: NLP

# Good documentation about tm can be found here:
# https://cran.r-project.org/web/packages/tm/vignettes/tm.pdf

# STEP 1: Raw text cleaning: case conversion
#(transform everything to lower case), remove
#punctuation and stopwords, and do a spelling check

# Load data
(reviews <- Corpus(VectorSource(reviews)))

#-# <<SimpleCorpus>>
#-# Metadata:  corpus specific: 1, document level (indexed): 0
#-# Content:  documents: 5

# It tells us that we have 86 reviews.
# VCorpus stands for Volatile Corpus (meaning it
# is stored in RAM (instead of stored on disk).
# Note that corpus refers to a collection of
# documents. It is the main structure for
# managing documents in the tm package.
# VectorSource means we are reading in a vector
# It also tells us that there is no metadata on
# the corpus level or document level.

# Let's look at the first document:
reviews[[1]]

#-# <<PlainTextDocument>>
#-# Metadata:  7
#-# Content:  chars: 385

# We see that there are 7 metadata fields
# These can be accessed using str(reviews[[1]])
# We also see the number of characters

#In what follows we try to reduce the number
#of unique words. We measure the mean number of
#words for a document in the corpus as follows:
unique_word_count <- function(data){
  if (any(class(data) %in% c("VCorpus", "Corpus"))) {
      data <- unlist(sapply(data,'[',"content"))
```

```
  }
  uniquewords <- unique(unlist(sapply(
                              strsplit(as.character(data)," "),
                              unique)))
  length(uniquewords)
}
unique_word_count(reviews)

#-# [1] 1

#Now we can apply the functions in the tm package:

#Case conversion transform all values to lower case
#Use mc.cores=1 to avoid any system specific problems
#The tm_map function is similar to the apply()
#function in that it applies a function to each
#element (document in this case).
#The function content_transformer allows to adapt
#the tolower() base function to work with the
# documents.
reviews <- tm_map(reviews,
                  content_transformer(tolower),
                  mc.cores=1)

#-# Error in FUN(content(x), ...): unused argument (mc.cores = 1)

unique_word_count(reviews)

#-# [1] 1

#The number of words decreased

#remove punctuation
reviews <- tm_map(reviews,
                  removePunctuation,
                  mc.cores=1)

#-# Error in FUN(content(x), ...): unused argument (mc.cores = 1)

unique_word_count(reviews)

#-# [1] 1

#The number of words decreased

#remove numbers
reviews <- tm_map(reviews,
                  removeNumbers,
                  mc.cores=1)

#-# Error in FUN(content(x), ...): unused argument (mc.cores = 1)
```

```
unique_word_count(reviews)

#-# [1] 1

#The number of words decreased

#remove stopwords
forremoval <- stopwords('english')
head(forremoval)

#-# [1] "i"      "me"     "my"     "myself" "we"     "our"

reviews <-  tm_map(reviews,
                   removeWords,
                   c(forremoval),
                   mc.cores=1)

#-# Error in FUN(content(x), ...): unused argument (mc.cores = 1)

unique_word_count(reviews)

#-# [1] 1

#The number of words decreased

#Collapse multiple white space in a
#single whitespace
reviews <-  tm_map(reviews,
                   stripWhitespace,
                   mc.cores=1)

#-# Error in FUN(content(x), ...): unused argument (mc.cores = 1)

#If we would want to look at the result,
#(e.g., the first document)
# we could use as.character()
# as.character((reviews[[1]]))

#Spell checking
#Based on Rasmus Baath, research blog,
#http://www.sumsar.net/blog/2014/12/peter-norvigs-spell-checker-in-two-lines-of-r/

#Download a list of words sorted by frequency of
#appearance in natural language

wordlist <-
  readLines("http://www.ballings.co/hidden/aCRM/data/chapter2/wordlist.txt")

#Function to correct misspelled words
correct <- function(word) {
  # How dissimilar is this word from all words in the wordlist?
  edit_dist <- adist(word, wordlist)
```

```
  # Is there a word that reasonably similar?
  # If yes, which ones?
  # If no, append the original word
  # Select the first result (because wordlist is sorted
  # from most common to least common)
  c(wordlist[edit_dist <= min(edit_dist,2)],word)[1]
}

#try out the function:
correct("speling")

#-# [1] "spelling"

correct("goodd")

#-# [1] "good"

correct("corect")

#-# [1] "correct"

correct("corrrect")

#-# [1] "correct"

correct("corrct")

#-# [1] "correct"

#Now convert the data back to a
#normal character vector. If we would look
#at a document using str we would see
#that it is a list with the actual text
#stored in the element "content".

reviews <- unlist(sapply(reviews,'[',"content"))

#Pre-allocate a vector where we will store the
#spell-checked reviews
reviews_spell_checked <- character(length(reviews))

#This loop takes a while:
start <- Sys.time()
for (i in 1:length(reviews)){
#Instead of applying correct() to each
#word, we fist make them unique
#The strsplit() function splits the string in words
count <- table(strsplit(reviews[i],' ')[[1]])
words <- names(count)

#Then we apply our correct function to each
```

```
# unique word
words <- as.character(sapply(words,correct))

#Next we reconstruct the original vector, but
#now spell-checked. We do this because we are
#going to exploit the tm package, which will
#count the words for us.
words <- rep(words,count)

#Concatenate back to a string
reviews_spell_checked[i] <-
  paste(words, collapse=" ")

#Print progress to the screen
if((i %% max(floor(length(reviews)/10),1))==0)
    cat(round((i/length(reviews))*100),"%\n")
}

#-# 20 %
#-# 40 %
#-# 60 %
#-# 80 %
#-# 100 %

Sys.time()-start

#-# Time difference of 0.002621174 secs

unique_word_count(reviews_spell_checked)

#-# [1] 0

# The word count is further reduced, but is
# it worth the long processing time?

# STEP 2: stemming

(reviews_spell_checked <- Corpus(VectorSource(reviews_spell_checked)))

#-# <<SimpleCorpus>>
#-# Metadata:  corpus specific: 1, document level (indexed): 0
#-# Content:  documents: 5

reviews_spell_checked <- tm_map(reviews_spell_checked,
                                stemDocument,
                                mc.cores=1)

#-# Error in FUN(content(x), ...): unused argument (mc.cores = 1)

unique_word_count(reviews_spell_checked)

#-# [1] 1
```

```
#The number of unique words is further reduced by stemming

# STEP 3, 4, 5: Create document-by-term matrix, Term filtering and
# Document vector weighting

#Only include words with a minimum length of 2 characters
(dtm_reviews <- DocumentTermMatrix(
          reviews_spell_checked,
          control = list(wordLengths = c(2, Inf),
                         weighting=function(x) weightTfIdf(x))
                                  )
)

#-# Warning in weightTfIdf(x): empty document(s): 1 2 3 4 5

#-# <<DocumentTermMatrix (documents: 5, terms: 0)>>
#-# Non-/sparse entries: 0/0
#-# Sparsity           : 100%
#-# Maximal term length: 0
#-# Weighting          : term frequency - inverse document frequency (normalized) (tf-idf)

#We see that the dtm is created. We also get the sparsity.

#Remove terms with too much sparsity. How much sparsity do we allow?
#Let's try different values for the sparse parameter

#Allow that 30% of the documents do not use a word
#At least 70% of the documents need to use the word for the
#word to stay in the dtm.
(dtm_reviews_dense <- removeSparseTerms(dtm_reviews, sparse= 0.3))

#-# <<DocumentTermMatrix (documents: 5, terms: 0)>>
#-# Non-/sparse entries: 0/0
#-# Sparsity           : 100%
#-# Maximal term length: 0
#-# Weighting          : term frequency - inverse document frequency (normalized) (tf-idf)

inspect(dtm_reviews_dense)

#-# <<DocumentTermMatrix (documents: 5, terms: 0)>>
#-# Non-/sparse entries: 0/0
#-# Sparsity           : 100%
#-# Maximal term length: 0
#-# Weighting          : term frequency - inverse document frequency (normalized) (tf-idf)
#-# Sample             :

#-# Warning in is.na(x): is.na() applied to non-(list or vector) of type 'NULL'
#-# Warning in is.na(j): is.na() applied to non-(list or vector) of type 'NULL'
#-# Error in `[.simple_triplet_matrix`(x, docs, terms): Invalid subscript type: NULL.

#Allow that 60% of the documents do not use a word
(dtm_reviews_dense <- removeSparseTerms(dtm_reviews, sparse= 0.6))
```

```
#-# <<DocumentTermMatrix (documents: 5, terms: 0)>>
#-# Non-/sparse entries: 0/0
#-# Sparsity           : 100%
#-# Maximal term length: 0
#-# Weighting          : term frequency - inverse document frequency (normalized) (tf-idf)

#Allow that 90% of the documents do not use a word
(dtm_reviews_dense <- removeSparseTerms(dtm_reviews, sparse= 0.9))

#-# <<DocumentTermMatrix (documents: 5, terms: 0)>>
#-# Non-/sparse entries: 0/0
#-# Sparsity           : 100%
#-# Maximal term length: 0
#-# Weighting          : term frequency - inverse document frequency (normalized) (tf-idf)

#There really is no way to know in advance how dense the dtm should be
#Therefore it is a reasonable approach not to remove too many sparse terms,
#and rely on the dimension reduction step that comes next.
```

Dimension reduction

In order to further reduce the number of columns in our document-by-term matrix (dtm) we use Latent Semantic Indexing (LSI). It works by grouping correlated terms. We will make linear combinations of terms that often appear together (Deerwester et al., 1990). Indexing in this case refers to creating a description or summary of documents. The words 'latent' and 'semantic' mean that there is some underlying partially obscured structure in the data. In other words, LSI groups observed terms (i.e., words) into latent concepts in order to describe documents more efficiently. Our approach to LSI is Singular Value Decomposition (SVD). The resulting concepts (or components) from SVD are orthogonal to each other (i.e., uncorrelated), and are called singular vectors. In addition they are ranked from high variance to low variance. To accomplish that SVD decomposes our *dtm* into three objects: *u*,*v*, *d*. In the following code chunk we show how to do SVD in *R*.

```
#Let's start from dtm_reviews_dense

reviews_mat <- as.matrix(dtm_reviews_dense)
dim(reviews_mat)

#-# [1] 5 0

#First center the data (it might help us when
# determining how many singular vectors we need to
# retain).
reviews_mat <- t(t(reviews_mat)-colMeans(reviews_mat))

#Perform svd
s <- svd(reviews_mat)
```

```
#-# Error in svd(reviews_mat): a dimension is zero

str(s)

#-# Error in str(s): object 's' not found

# u is the document-by-concept matrix
# This is the matrix we are interested in
dim(s$u)

#-# Error in eval(expr, envir, enclos): object 's' not found

#ncol(s$u) will be equal to ncol(reviews_mat)
#if nrow(reviews_mat) >= ncol(reviews_mat),
#otherwise ncol(s$u) will be equal to nrow(reviews_mat)

#d represents the strength of each concept
length(s$d)

#-# Error in eval(expr, envir, enclos): object 's' not found

#v is the term-to-concept matrix, it shows how the
# terms are related to the concepts
dim(s$v)

#-# Error in eval(expr, envir, enclos): object 's' not found

#We will use u in our basetable.
head(s$u)

#-# Error in head(s$u): object 's' not found

#However, it is important to note that we will want
#to deploy our our model on future data. We can do
#that as follows:
head(reviews_mat %*% s$v  %*% solve(diag(s$d)))

#-# Error in head(reviews_mat %*% s$v %*% solve(diag(s$d))): object 's' not found

#The only thing we need to do is replace reviews_mat
#with the new dtm.
#Note: solve returns the inverse of diag(s$d),
#it is equivalent to 1/sv$d.

# d is proportional to the variance that is explained
# To compute the variance we proceed as follows:
(s$d^2)/(nrow(reviews_mat) - 1)

#-# Error in eval(expr, envir, enclos): object 's' not found

# Finally we can plot the explained variance per
# singular vector in a scree plot.
# % Variance explained of total variance by svd
plot(s$d^2/sum(s$d^2),
     type="b",
     ylab="% variance explained",
     xlab="Singular vectors",
     xaxt="n"
)
```

```
#-# Error in plot(s$d^2/sum(s$d^2), type = "b", ylab = "% variance explained", : object 's' not
found

axis(1,at=1:length(s$d^2/sum(s$d^2)))

#-# Error in axis(1, at = 1:length(s$d^2/sum(s$d^2))): object 's' not found

#The first singular vector explains 44.8% of the variance
#The first two singular vectors explain 90.8% of the variance.
#We could drop the last two without losing much information.

# This shows that we can go from 171 columns
# (ncol(reviews_mat)) to two without loosing much information.
```

It is important to realize that the result of this entire process is just matrix with concepts as variables. We can use them as normal predictors.

2.4.3.12 Speeding up code with the data.table package

The `data.table` package will speed up our code considerably. It is also more memory efficient because of passing-by-reference instead passing-by-value (i.e., copying). Moreover it allows shortcuts in our code, so we will have to do less typing. This makes our code easier to develop and maintain. However, we will need some time getting used to the different syntax. In the following code chunk we will cover the key programming statements.

```
#Install and load the data.table package
if (!require(data.table)) {
  install.packages("data.table")
  require(data.table)
}

#-# Loading required package: data.table

# Consider the following data frame
(df <- data.frame(V1=c("a","b","a","a","b"),
                  V2=1:5,
                  V3=c(6,6,7,7,8),
                  stringsAsFactors=FALSE))

#-#   V1 V2 V3
#-# 1  a  1  6
#-# 2  b  2  6
#-# 3  a  3  7
#-# 4  a  4  7
#-# 5  b  5  8

# Now, consider the same data frame, but as a data table
(dt <- data.table(V1=c("a","b","a","a","b"),
                  V2=1:5,
                  V3=c(6,6,7,7,8)))
```

```
#-#    V1 V2 V3
#-# 1:  a  1  6
#-# 2:  b  2  6
#-# 3:  a  3  7
#-# 4:  a  4  7
#-# 5:  b  5  8

#note the difference in notation:
#the data table has colons next to the row names

# We can easily convert data frames to data tables.
(dt2 <- data.table(df))

#-#    V1 V2 V3
#-# 1:  a  1  6
#-# 2:  b  2  6
#-# 3:  a  3  7
#-# 4:  a  4  7
#-# 5:  b  5  8

#We can see all data tables in memory using tables()
#tables() is unrelated to table()
tables()

#-#      NAME NROW NCOL MB COLS     KEY
#-# [1,] dt      5    3  1 V1,V2,V3
#-# [2,] dt2     5    3  1 V1,V2,V3
#-# Total: 2MB

# You may have noticed the empty column called KEY
# Setting a key (keying; indexing) can speed up operations on the data table.
# Think of indexing in terms of a phone book. A phone book
# is indexed by name. Finding the phone number for a person
# if very easy. However, finding the name for a phone number
# is very hard, if not impossible. Indexing a data table
# is equivalent to sorting, and is out of the scope
# of this book.

# Let's look at the class of a data.table
class(dt)

#-# [1] "data.table" "data.frame"

class(df)

#-# [1] "data.frame"

# Note that a data table also is of class data.frame.
# However, do not assume that code that would run
# on a data frame would run on a data.table.
# You need very different code to interact with
```

```
# a data.table.

#########################################################
# Subsetting a data.table (dt) vs a data.frame (df)
#########################################################
df

#-#   V1 V2 V3
#-# 1  a  1  6
#-# 2  b  2  6
#-# 3  a  3  7
#-# 4  a  4  7
#-# 5  b  5  8

dt

#-#    V1 V2 V3
#-# 1:  a  1  6
#-# 2:  b  2  6
#-# 3:  a  3  7
#-# 4:  a  4  7
#-# 5:  b  5  8

######################
# by $-sign
df$V1

#-# [1] "a" "b" "a" "a" "b"

dt$V1 #note that dimensions are dropped just like in a df

#-# [1] "a" "b" "a" "a" "b"

df[df$V1=="a",]

#-#   V1 V2 V3
#-# 1  a  1  6
#-# 3  a  3  7
#-# 4  a  4  7

dt[V1=="a",]

#-#    V1 V2 V3
#-# 1:  a  1  6
#-# 2:  a  3  7
#-# 3:  a  4  7

######################
# by name
df[, "V1"]
```

```
#-# [1] "a" "b" "a" "a" "b"

with(df,V1)

#-# [1] "a" "b" "a" "a" "b"

dt[, V1]

#-# [1] "a" "b" "a" "a" "b"

dt[, .(V1)] #equivalent to drop=FALSE in df

#-#    V1
#-# 1:  a
#-# 2:  b
#-# 3:  a
#-# 4:  a
#-# 5:  b

dt[, list(V1)] #equivalent to drop=FALSE in df, note that . is short for list

#-#    V1
#-# 1:  a
#-# 2:  b
#-# 3:  a
#-# 4:  a
#-# 5:  b

dt[, "V1"]

#-#    V1
#-# 1:  a
#-# 2:  b
#-# 3:  a
#-# 4:  a
#-# 5:  b

# In ealier versions we had to set with=FALSE,
# but now it sets with=FALSE automatically when there are no unquoted variable names:
dt[, "V1",  with=FALSE]

#-#    V1
#-# 1:  a
#-# 2:  b
#-# 3:  a
#-# 4:  a
#-# 5:  b

dt[["V1"]] #note that dimensions are dropped as opposed to dt[, "V1"]

#-# [1] "a" "b" "a" "a" "b"
```

```
df[,c("V1","V2")]

#-#   V1 V2
#-# 1  a  1
#-# 2  b  2
#-# 3  a  3
#-# 4  a  4
#-# 5  b  5

dt[,c("V1","V2")]

#-#    V1 V2
#-# 1:  a  1
#-# 2:  b  2
#-# 3:  a  3
#-# 4:  a  4
#-# 5:  b  5

# In ealier versions we had to set with=FALSE,
# but now it sets with=FALSE automatically when there are no unquoted variable names
dt[,c("V1","V2"), with=FALSE]

#-#    V1 V2
#-# 1:  a  1
#-# 2:  b  2
#-# 3:  a  3
#-# 4:  a  4
#-# 5:  b  5

dt[,.(V1,V2)]

#-#    V1 V2
#-# 1:  a  1
#-# 2:  b  2
#-# 3:  a  3
#-# 4:  a  4
#-# 5:  b  5

dt[,list(V1,V2)]

#-#    V1 V2
#-# 1:  a  1
#-# 2:  b  2
#-# 3:  a  3
#-# 4:  a  4
#-# 5:  b  5

######################
# by indices

# for columns:
df[,1]
```

```
#-# [1] "a" "b" "a" "a" "b"

df[,1, drop=FALSE]

#-#   V1
#-# 1  a
#-# 2  b
#-# 3  a
#-# 4  a
#-# 5  b

df[,1:2]

#-#   V1 V2
#-# 1  a  1
#-# 2  b  2
#-# 3  a  3
#-# 4  a  4
#-# 5  b  5

df[,c(1:2)]

#-#   V1 V2
#-# 1  a  1
#-# 2  b  2
#-# 3  a  3
#-# 4  a  4
#-# 5  b  5

dt[,1]

#-#    V1
#-# 1:  a
#-# 2:  b
#-# 3:  a
#-# 4:  a
#-# 5:  b

# In ealier versions we had to set with=FALSE,
# but now it sets with=FALSE automatically when there are no unquoted variable names
dt[,1, with=FALSE]

#-#    V1
#-# 1:  a
#-# 2:  b
#-# 3:  a
#-# 4:  a
#-# 5:  b

dt[,1:2]
```

```
#-#    V1 V2
#-# 1:  a  1
#-# 2:  b  2
#-# 3:  a  3
#-# 4:  a  4
#-# 5:  b  5

dt[,c(1:2)]

#-#    V1 V2
#-# 1:  a  1
#-# 2:  b  2
#-# 3:  a  3
#-# 4:  a  4
#-# 5:  b  5

df[1]

#-#   V1
#-# 1  a
#-# 2  b
#-# 3  a
#-# 4  a
#-# 5  b

df[[1]]

#-# [1] "a" "b" "a" "a" "b"

#Note that dimensions are dropped like in dt[, V1] as
#opposed to dt[,1] and dt[, "V1"]:
dt[[1]]

#-# [1] "a" "b" "a" "a" "b"

# for rows
df[1:3,]

#-#   V1 V2 V3
#-# 1  a  1  6
#-# 2  b  2  6
#-# 3  a  3  7

dt[1:3,]

#-#    V1 V2 V3
#-# 1:  a  1  6
#-# 2:  b  2  6
#-# 3:  a  3  7
```

```
#########################################################
# Aggregating a data.table (dt) vs a data.frame (df)
#########################################################

# data.table syntax borrows from SQL as follows:
# dt[i,j,by]
# R:    i           j                by
# SQL:  where       select/update    group by

# Equivalent SQL syntax:
# SELECT j
# FROM dt
# WHERE i
# GROUP BY: by

# one by variable
aggregate(df$V2,by=list(V1=df$V1),sum)

#-#   V1 x
#-# 1  a 8
#-# 2  b 7

dt[,sum(V2),by=V1]

#-#    V1 V1
#-# 1:  a  8
#-# 2:  b  7

# multiple aggregation variables
aggregate(df[,c("V2","V3")],by=list(V1=df$V1),sum)

#-#   V1 V2 V3
#-# 1  a  8 20
#-# 2  b  7 14

dt[,.(sum(V2),sum(V3)),by=V1]

#-#    V1 V1 V2
#-# 1:  a  8 20
#-# 2:  b  7 14

# multiple by variables
aggregate(df[,"V2"],by=list(V1=df$V1,V3=df$V3),sum)

#-#   V1 V3 x
#-# 1  a  6 1
#-# 2  b  6 2
#-# 3  a  7 7
#-# 4  b  8 5

dt[,.(V2=sum(V2)),by=.(V1,V3)]
```

```
#-#    V1 V3 V2
#-# 1:  a  6  1
#-# 2:  b  6  2
#-# 3:  a  7  7
#-# 4:  b  8  5

##########################################################
# Creating new variables in a data.table (dt) vs
# a data.frame (df)
##########################################################

# Creating new variables
df$V4 <- df$V2 + df$V3
dt$V4 <- df$V2 + df$V3

# or better, by reference:
dt

#-#    V1 V2 V3 V4
#-# 1:  a  1  6  7
#-# 2:  b  2  6  8
#-# 3:  a  3  7 10
#-# 4:  a  4  7 11
#-# 5:  b  5  8 13

dt[ ,V5 := V2 + V3 ]
dt

#-#    V1 V2 V3 V4 V5
#-# 1:  a  1  6  7  7
#-# 2:  b  2  6  8  8
#-# 3:  a  3  7 10 10
#-# 4:  a  4  7 11 11
#-# 5:  b  5  8 13 13

# Note that there is no '<-' involved; we are creating a new variable
# by reference. This is a lot more efficient. If we use the '<-' we
# are copying the entire dt. If we use ':=' we are not.

# Removing variables
df

#-#   V1 V2 V3 V4
#-# 1  a  1  6  7
#-# 2  b  2  6  8
#-# 3  a  3  7 10
#-# 4  a  4  7 11
#-# 5  b  5  8 13

df$V4 <- NULL
df
```

```
#-#   V1 V2 V3
#-# 1  a  1  6
#-# 2  b  2  6
#-# 3  a  3  7
#-# 4  a  4  7
#-# 5  b  5  8

dt

#-#    V1 V2 V3 V4 V5
#-# 1:  a  1  6  7  7
#-# 2:  b  2  6  8  8
#-# 3:  a  3  7 10 10
#-# 4:  a  4  7 11 11
#-# 5:  b  5  8 13 13

dt[, V4 := NULL]
dt

#-#    V1 V2 V3 V5
#-# 1:  a  1  6  7
#-# 2:  b  2  6  8
#-# 3:  a  3  7 10
#-# 4:  a  4  7 11
#-# 5:  b  5  8 13

#########################################################
#  Merging data tables (dt) vs data frames (df)
#########################################################

# data frame
(df1 <- data.frame(V1=c("a","b"),V2=1:2,V3=c(6,6)))

#-#   V1 V2 V3
#-# 1  a  1  6
#-# 2  b  2  6

(df2 <- data.frame(V1=c("b","c"),V4=3:4,V5=c(7,8)))

#-#   V1 V4 V5
#-# 1  b  3  7
#-# 2  c  4  8

merge(df1,df2,by="V1")

#-#   V1 V2 V3 V4 V5
#-# 1  b  2  6  3  7

merge(df1,df2,by="V1", all=TRUE)
```

```
#-#   V1 V2 V3 V4 V5
#-# 1  a  1  6 NA NA
#-# 2  b  2  6  3  7
#-# 3  c NA NA  4  8

# data table
(dt1 <- data.table(V1=c("a","b"),V2=1:2,V3=c(6,6)))

#-#    V1 V2 V3
#-# 1:  a  1  6
#-# 2:  b  2  6

(dt2 <- data.table(V1=c("b","c"),V4=3:4,V5=c(7,8)))

#-#    V1 V4 V5
#-# 1:  b  3  7
#-# 2:  c  4  8

merge(dt1,dt2,by="V1")

#-#    V1 V2 V3 V4 V5
#-# 1:  b  2  6  3  7

merge(dt1,dt2,by="V1", all=TRUE)

#-#    V1 V2 V3 V4 V5
#-# 1:  a  1  6 NA NA
#-# 2:  b  2  6  3  7
#-# 3:  c NA NA  4  8

##########################################################
#  Reading in data as a data.table (dt) vs a
#  data.frame (df)
##########################################################

URL_subs <-
  "http://ballings.co/hidden/aCRM/data/chapter2/subscriptions.txt"

#the usual way
subs1  <- read.table(URL_subs,
                     header=TRUE,
                     sep=";")

# Using the data.table package
# If you have a large data set stored on disk,
# fread will be lightning fast compared to read.table
subs2  <- fread(URL_subs,
                header=TRUE,
                sep=";")

##########################################################
```

```
# rbinding elements in a list as a data.table (dt) vs
# a data.frame (df)
#########################################################

# data frame
df1 <- data.frame(a=c(1,2),b=c(3,4),c=c(5,6))
df2 <- data.frame(a=c(7,8),b=c(9,10),c=c(11,12))

(l <- list(df1,df2))

#-# [[1]]
#-#   a b c
#-# 1 1 3 5
#-# 2 2 4 6
#-#
#-# [[2]]
#-#   a  b  c
#-# 1 7  9 11
#-# 2 8 10 12

do.call(rbind,l)

#-#   a  b  c
#-# 1 1  3  5
#-# 2 2  4  6
#-# 3 7  9 11
#-# 4 8 10 12

# data table
dt1 <- data.table(a=c(1,2),b=c(3,4),c=c(5,6))
dt2 <- data.table(a=c(7,8),b=c(9,10),c=c(11,12))

(l <- list(dt1,dt2))

#-# [[1]]
#-#    a b c
#-# 1: 1 3 5
#-# 2: 2 4 6
#-#
#-# [[2]]
#-#    a  b  c
#-# 1: 7  9 11
#-# 2: 8 10 12

rbindlist(l)

#-#    a  b  c
#-# 1: 1  3  5
#-# 2: 2  4  6
#-# 3: 7  9 11
#-# 4: 8 10 12
```

2.4.4 Guidelines for basetable creation

To start building a basetable there are two important questions that need to be answered; (1) 'Which customers should be included in the basetable?' and (2) 'How to compute the response (i.e., dependent) variable?'. The answers to these questions differ based on which CRM application (acquisition, up-sell, cross-sell, churn and customer lifetime value) we are working. In this section we will provide a short overview that will come in handy when working on the applications.

Recall that t1 stands for the start of the independent period, t2 stands for the end of the independent period, t3 stands for the start of the dependent period, and t4 stands for the end of the dependent period. More information about the time window is available in Section 2.4.1. Independent variables are based on data between t1 and t2. The response variable is computed on data between t3 and t4.

2.4.4.1 Which customers to include?

Acquisition
Include only those customers that were not a customer at t2 (i.e., prospects). In addition we also need all the prospects that did not become customers between t3 and t4.

Up-sell
The customer is active (has bought recently, or has a subscription) at t2, and has at least one unit of the product that we want to sell more of.

Cross-sell
The customer made a transaction in [t3,t4].

Churn
The customer is active at t2.

Customer Lifetime Value, CLV

If we assume that a lost customer can never come back then only customers that are active at t2 are included. This setting is called the 'lost-for- good' setting. If we assume that customers can come back (i.e., inactivity is only temporary) then we include lost and active customers. This setting is called 'always-a-share' (Dwyer, 1997). Lost-for-good settings are often characterized by the involvement of significant resource commitment by the customer (e.g., contracts such as financial services, magazine subscriptions). Retail is an example of always-a-share setting.

In practice, management decides whether it wants a lost-for-good or an always-a-share CLV. The choice is often driven by the underlying strategy that would require different communication strategies depending on whether the customer is active or not (e.g., it would be inapproriate to thank an inactive customer for his or her business with the company). That is a common reason why management often prefers a lost-for-good approach.

From a modeling standpoint always-a-share can also present considerable challenges. In always-a-share models all customers, active and inactive ones, are kept in the basetable. This often results in very good models overall because a large number of customers has become inactive over the

years and many have not come back. This means that one independent variable, recency of purchase, becomes an extremely strong predictor in differentiating customers with a value of 0 for the dependent variable from those with a value bigger than 0 for the dependent variable. For these reasons one will often see a lost-for-good approach or an always-a-share model with additional inclusion requirements for customers (e.g., customers must have purchase in the last 90 days).

2.4.4.2 How to compute the response variable?

Acquisition

$$acquisition = \begin{cases} 1, & \text{if prospect becomes customer in [t3,t4].} \\ 0, & \text{otherwise.} \end{cases} \tag{2.1}$$

Up-sell

$$up - sell = \begin{cases} 1, & \text{if customer buys more in [t3,t4].} \\ 0, & \text{otherwise.} \end{cases} \tag{2.2}$$

Cross-sell

$$cross - sell = \begin{cases} 1, & \text{if customer buys product x in [t3,t4].} \\ 2, & \text{if customer buys product y in [t3,t4].} \\ 3, & \text{if customer buys product z in [t3,t4].} \\ \ldots, & \ldots. \end{cases} \tag{2.3}$$

Churn

$$churn = \begin{cases} 1, & \text{if customer stops buying in [t3,t4].} \\ 0, & \text{otherwise.} \end{cases} \tag{2.4}$$

Customer Lifetime Value

There are many definitions of CLV, but we will only use one, the most general and widely applicable one. We define CLV for a customer i for a finite time horizon (Berger and Nasr, 1998) with irregular cash flows as follows:

$$CLV_i = \sum_{p=1}^{P} \frac{Profit_{i,p}}{(1+r)^{\frac{d_p - d_0}{365}}} \tag{2.5}$$

where P is the total number of purchases, r is the discount rate, d_p is the date of a given purchase incidence, and d_0 is the start of the dependent period (t3). The profit of a customer i of a given

purchase p is defined as the price times a margin. If different products are involved, and the margin is unknown for the different products, profit is replaced by revenue. If only one product is involved it makes no difference for ranking if revenue or profit is used. An often used discount rate r is 4%. A common length of the dependent period is 3 years.

2.5 Modeling

Once the basetable is ready for analysis (one row per customer or prospect, one dependent variable) we need to select an algorithm to create a model. Wolpert's no free lunch theorem states that no algorithm works best on all datasets (Wolpert, 1996; Wolpert and Macready, 1997). Hence models based on several algorithms will need to be estimated and evaluated so we can select the best one. In this section we will cover naive bayes, logistic regression, neural networks, k-nearest neighbors, decision trees, bagging, random forest, kernel factory, adaptive boosting, rotation forest, and support vector machines.

For binary classification we will use one performance measure in this section called Area Under the Receiver Operating Characteristics curve (AUC). AUC is a measure between 0.5 (if the model is not doing better than random selection) and 1 (if the model makes perfect predictions). It is possible that the AUC is smaller than 0.5 and this always indicates overfitting. At this point this is all we need to know. In Section 2.6 we will provide details about the AUC measure and other measures.

2.5.1 Training, validation, and test set

Once we have prepared the basetable we need to partition the data in a training set (to estimate the model, possibly multiple times using different parameters), validation set (to make predictions, evaluate models, and pick the best model; also called tuning), and a test set (to evaluate the model that we selected based on validation set performance one more time to obtain the final model performance). If the algorithm that we use does not require tuning we don't need a validation set. If we compare an algorithm that needs tuning with an algorithm that does not require tuning it is only fair that in the latter case we estimate the model on the combined set of what would otherwise be a training set and validation set. In short when the algorithm does not require tuning we use the training plus validation set to estimate the model.

Why do we train and test on different data sets? We want to make sure we are not overfitting the data (learning random error or noise). Why do we need a validation and test set? If we determined final performance of the best model on the validation set, the best model might just as well be lucky on this particular data set. We want to make sure that if we use the model on a future data set, we get optimal results. Therefore final performance can only be determined on a data set that has not been used during any training or tuning; a test set.
All code in this section is available at:
`http://ballings.co/hidden/aCRM/code/chapter2/TrainValTest.R`

```
classes <- c("character",
             "numeric",
             "numeric",
             "numeric",
             "numeric",
             "numeric",
             "numeric",
             "numeric",
             "factor")

Basetable <- read.csv("http://ballings.co/hidden/aCRM/data/chapter2/Basetable.csv",
                      colClasses=classes)
#look at the data
str(Basetable, vec.len=0.5)

#-# 'data.frame': 2122 obs. of  9 variables:
#-#  $ CustomerID            : chr   ...
#-#  $ TotalDiscount         : num  0 ...
#-#  $ TotalPrice            : num  962 ...
#-#  $ TotalCredit           : num  -10.1 ...
#-#  $ PaymentType_DD        : num  0 ...
#-#  $ PaymentStatus_Not.Paid: num  0 ...
#-#  $ Frequency             : num  4 ...
#-#  $ Recency               : num  193 ...
#-#  $ Churn                 : Factor w/ 2 levels "0","1": 1 ...

#We don't need the customer ID in the modeling phase
#so we can safely remove it
Basetable$CustomerID <- NULL

#Before starting the analyses, clean up by removing all
# objects except the Basetable
rm(list=setdiff(ls(),c("Basetable","classes")))
#setdiff returns which elements of the first argument
#are not in the second argument
ls()

#-# [1] "Basetable" "classes"

#Partition the data in three: train, val and test.

#create idicators
#randomize order of indicators
allind <- sample(x=1:nrow(Basetable),size=nrow(Basetable))
#split in three parts
trainind <- allind[1:round(length(allind)/3)]
valind <- allind[(round(length(allind)/3)+1):round(length(allind)*(2/3))]
testind <- allind[round(length(allind)*(2/3)+1):length(allind)]

BasetableTRAIN <- Basetable[trainind,]
BasetableVAL <- Basetable[valind,]
```

```
BasetableTEST <- Basetable[testind,]
#if no tuning is required than we
#use TRAIN + VAL as training set
BasetableTRAINbig <- rbind(BasetableTRAIN,BasetableVAL)

#To make sure that the results in the book match our
#results as closely as possible it's better to work on these
#preprepared sets:
BasetableTRAIN <-
  read.csv("http://ballings.co/hidden/aCRM/data/chapter2/BasetableTRAIN.csv",
                      colClasses=classes)
BasetableVAL <-
  read.csv("http://ballings.co/hidden/aCRM/data/chapter2/BasetableVAL.csv",
                      colClasses=classes)
BasetableTEST <-
  read.csv("http://ballings.co/hidden/aCRM/data/chapter2/BasetableTEST.csv",
                      colClasses=classes)
BasetableTRAINbig <-
  read.csv("http://ballings.co/hidden/aCRM/data/chapter2/BasetableTRAINbig.csv",
                      colClasses=classes)

#At this point we can remove the Basetable
rm(Basetable)

BasetableTRAIN$CustomerID <- BasetableVAL$CustomerID <- NULL
BasetableTEST$CustomerID <- BasetableTRAINbig$CustomerID <- NULL

#Isolate the response variable
#This makes it easier to call some functions later on
yTRAIN <- BasetableTRAIN$Churn
BasetableTRAIN$Churn <- NULL

yVAL <- BasetableVAL$Churn
BasetableVAL$Churn <- NULL

yTEST <- BasetableTEST$Churn
BasetableTEST$Churn <- NULL

yTRAINbig <- BasetableTRAINbig$Churn
BasetableTRAINbig$Churn <- NULL

#Check the distribution of the dependent variable.
#It should be similar in the three sets.
table(yTRAIN);table(yVAL);table(yTEST)


#-# yTRAIN
#-#   0   1
#-# 671  36
#-# yVAL
#-#   0   1
#-# 663  45
```

```
#-# yTEST
#-#   0   1
#-# 669  38

#check whether we didn't make a mistake
dim(BasetableTRAIN)

#-# [1] 707   7

dim(BasetableVAL)

#-# [1] 708   7

dim(BasetableTEST)

#-# [1] 707   7
```

How will we use these datasets? Suppose we have an algorithm with a parameter that has 10 possible values. We'll train 10 different models, each with a different value for the parameter on the training set. Then we assess the performance of these 10 models on the validation set and select the best model. To obtain the final performance of the selected model we assess performance on the test set.

2.5.2 Building blocks of algorithms

Machine learning algorithms, also called learners, consist of combinations of three components: representation, evaluation and optimization (Domingos, 2012). Representation stands for the structure of the learner (e.g., networks, trees), evaluation represents the objective function (e.g., accuracy, likelihood, squared error, information gain), and optimization denotes the method to search in a given representation for the highest performance of the objective function (e.g., greedy search, branch-and-bound, gradient descent). A learner inputs a data set of instances (x_i,y_i), also called patterns, objects or observations, with $(i = 1, 2, 3, \ldots, N)$, where $x_i = (x_{i1}, \ldots, x_{in}) \in \mathbb{R}^n$ is the feature vector of instance i, $y_i \in \{0,1\}$ is the class label of instance i, n is the dimensionality of the input space, and N is the number of instances. The goal of a learner is to output a classifier, also called hypothesis or classification model.

2.5.3 Naive bayes

Bayes's theorem (alternatively Bayes' rule or Bayes' law), named after Thomas Bayes (1701-1761), is defined as follows for a binary dependent variable:

$$P(Y = 1 \mid X = k) = \frac{P(X = k \mid Y = 1)\, P(Y = 1)}{P(X = k)} \tag{2.6}$$

where *Y={0,1}*, *X={0,1,...,k}*, *P(Y=1)* and *P(X=k)* are the probabilities of *Y=1* and *X=k*, *P(X=k | Y=1)* is the probability of observing event *X=k* given *Y=1* is true, *P(Y=1 | X=k)* is

the probability of observing event *Y=1* given *X=k* is true, and $P(X = k) = \sum_{y \in Y} P(X = k \mid Y_y)P(Y_y)$. In case of a binary *Y={0,1}*, $P(X = k) = P(X = k \mid Y = 0)P(Y = 0) + P(X = k \mid Y = 1)P(Y = 1)$. Note that we use *X* for independent predictors and *Y* for the dependent variables.

P(Y=1) is the prior probability of *Y=1* because it does not take into account any information about *X*. *P(X=k)* is the prior probability of *X=k* because it does not take into account any information about *Y=1* and is also called the normalizing constant. *P(Y=1 | X=k)* is the posterior probability because it depends on *X=k*. *P(X=k | Y=1)* is the posterior probability because it depends on *Y=1* and is also called the likelihood. As an equation:

$$posterior = \frac{likelihood \times prior}{normalizing\ constant} \tag{2.7}$$

For example, consider *X*={complaint, no complaint} and *Y*={churn, no churn}. If the probability that a customer has made a complaint in the past (*X=complaint*), when he or she churns in the future (*Y=churn*) is *P(X=complaint | Y=churn)*=20%, the probability of churn is *P(Y=churn)*=10%, and the probability of a complaint is *P(X=complaint)*=5%, then *P(Y=churn | X=complaint)*=0.4 (0.2*0.1/0.05).

Now how does this work if we have more than one predictor (*X*)? In the other algorithms in this book multiple *X* will be treated simultaneously (i.e., they are dependent). In naive bayes all predictors are assumed to be independent. For example, the variables 'length of relationship' (tenure), and total spend of a customer are likely to be correlated. In naive bayes they are treated as independent, hence the name 'naive'. Bayes' rule without the independence assumption is called a bayesian network. With the independence assumption we can write Bayes' rule for *q* predictors as follows:

$$P(Y = 1 \mid (X_1 = k_1 \ \& \ X_2 = k_2 \ \& \ \ldots \ \& \ X_q = k_q)) = \frac{P(Y = 1) \prod_{q=1}^{Q} P(X_q = k_q \mid Y = 1)}{P(Y = 0) \prod_{q=1}^{Q} P(X_q = k_q \mid Y = 0) + P(Y = 1) \prod_{q=1}^{Q} P(X_q = k_q \mid Y = 1)} \tag{2.8}$$

where *Q* denotes the number of variables, and k_q denotes a category of variable q. To understand naive bayes, let's look at a concrete example. This code chunk is available from: `http://ballings.co/hidden/aCRM/code/chapter2/NaiveBayes.R`

```
# Consider the following toy dataset
(df <-  data.frame( churn= c(0,0,1,1,1,0,1,0,1,1,1,1,1,0),
                    tenure=c("long","long","short","medium",
                             "medium","medium","short","long",
                             "long","medium","long","short",
                             "short","medium"),
                    spend= c("high","high","high","medium",
                             "low","low","low","medium",
                             "low","medium","medium","medium",
                             "high","medium")))
```

```
#-#    churn tenure  spend
#-# 1      0   long   high
#-# 2      0   long   high
#-# 3      1  short   high
#-# 4      1 medium medium
#-# 5      1 medium    low
#-# 6      0 medium    low
#-# 7      1  short    low
#-# 8      0   long medium
#-# 9      1   long    low
#-# 10     1 medium medium
#-# 11     1   long medium
#-# 12     1  short medium
#-# 13     1  short   high
#-# 14     0 medium medium

# We're going to compute all the terms that we need
# in our equation to compute the probability of churn
# for a given row.

# churn: compute the percentage
table(df[,1])

#-#
#-# 0 1
#-# 5 9

table(df[,1])/sum(table(df[,1]))

#-#
#-#         0         1
#-# 0.3571429 0.6428571

#tenure: compute the percentage by column total
table(df[,2],df[,1])

#-#
#-#          0 1
#-#   long   3 2
#-#   medium 2 3
#-#   short  0 4

t(t(table(df[,2],df[,1]))/colSums(table(df[,2],df[,1])))

#-#
#-#                  0         1
#-#   long   0.6000000 0.2222222
#-#   medium 0.4000000 0.3333333
#-#   short  0.0000000 0.4444444

#spend: compute the percentage by column total
table(df[,3],df[,1])
```

```
#-#
#-#          0 1
#-#   high   2 2
#-#   low    1 3
#-#   medium 2 4

t(t(table(df[,3],df[,1]))/colSums(table(df[,3],df[,1])))

#-#
#-#                  0         1
#-#   high   0.4000000 0.2222222
#-#   low    0.2000000 0.3333333
#-#   medium 0.4000000 0.4444444

# The above contingency tables are the
# inputs for our model.

# Consider a new instance with the following characteristics:
#   tenure=long
#   spend=high


# What is the probability of churn?

# P (churn=1 | Xtenure= long & Xspend= high) =

# Numerator:
#       P( Xtenure=long | churn = 1 )
#     x P( Xspend=high | churn = 1 )
#     x P( churn = 1) =
(Numerator <- 0.2222222 * 0.2222222 * 0.6428571)

#-# [1] 0.03174602

# Denominator:
#     P ( Xtenure= long & Xspend= high )=
#     P( Xtenure=long | churn = 0 )
#     x P( Xspend=high | churn = 0 )
#     x P( churn = 0)
#   +   P( Xtenure=long | churn = 1 )
#     x P( Xspend=high | churn = 1 )
#     x P( churn = 1)
(Denominator <-
(0.6000000 * 0.4000000 * 0.3571429) +
(0.2222222 * 0.2222222 * 0.6428571))

#-# [1] 0.1174603

Numerator/Denominator

#-# [1] 0.2702702
```

```
# Let's verify this with the naiveBayes
# function in the e1071 package

if (!require("e1071")) {
  install.packages('e1071',
                   repos="https://cran.rstudio.com/",
                   quiet=TRUE)
  require('e1071')
}

#-# Loading required package: e1071
#-#
#-# Attaching package: 'e1071'
#-# The following object is masked from 'package:imputeMissings':
#-#
#-#    impute

head(df, n=1)

#-#   churn tenure spend
#-# 1     0   long  high

NB <- naiveBayes(x=df[,2:3], y=df[,1])
(predNB <- predict(NB,df[1,2:3], type = "raw", threshold = 0.001)[,2])

#-#         1
#-# 0.2702703

# As we can see, the first row in df is
# equivalent to the example we computed manually

#It is important to note that categorical variables need to be
#factors with the same levels in both training and new data.
#This can be problematic in a predictive context if the new data
#obtained in the future does not contain all the levels.
#The solution is to map the levels of the training data to
#the new data

#Suppose this is the training data
df

#-#    churn tenure  spend
#-# 1      0   long   high
#-# 2      0   long   high
#-# 3      1  short   high
#-# 4      1 medium medium
#-# 5      1 medium    low
#-# 6      0 medium    low
#-# 7      1  short    low
#-# 8      0   long medium
#-# 9      1   long    low
```

```
#-# 10     1 medium medium
#-# 11     1   long medium
#-# 12     1  short medium
#-# 13     1  short   high
#-# 14     0 medium medium

str(df)

#-# 'data.frame': 14 obs. of  3 variables:
#-#  $ churn : num  0 0 1 1 1 0 1 0 1 1 ...
#-#  $ tenure: Factor w/ 3 levels "long","medium",..: 1 1 3 2 2 2 3 1 1 2 ...
#-#  $ spend : Factor w/ 3 levels "high","low","medium": 1 1 1 3 2 2 2 3 2 3 ...

#And this is the new data
(new  <- data.frame(tenure="short",
                    spend="high"))

#-#   tenure spend
#-# 1  short  high

str(new)

#-# 'data.frame': 1 obs. of  2 variables:
#-#  $ tenure: Factor w/ 1 level "short": 1
#-#  $ spend : Factor w/ 1 level "high": 1

#Let's fit the model
(predNB <- predict(NB,
                   new,
                   type = "raw",
                   threshold = 0.001))[,2]

#-#         1
#-# 0.2702703

#This value is actually incorrect

#What we really need to do is create new data with the same factor levels
(new <- data.frame(tenure=factor("short", levels=c("long","medium","short")),
                   spend=factor("high",levels=c("high","low","medium"))))

#-#   tenure spend
#-# 1  short  high

str(new)

#-# 'data.frame': 1 obs. of  2 variables:
#-#  $ tenure: Factor w/ 3 levels "long","medium",..: 3
#-#  $ spend : Factor w/ 3 levels "high","low","medium": 1
```

```
(predNB <- predict(NB,
                   new,
                   type = "raw",
                   threshold = 0.001))[,2]

#-#         1
#-# 0.9977551

#This value is correct

#How to do this automatically?
# Extract the levels from the training data
(lev <- sapply(df[,2:3],levels))

#-#      tenure   spend
#-# [1,] "long"   "high"
#-# [2,] "medium" "low"
#-# [3,] "short"  "medium"

#Then apply the levels to the new data
(new  <- data.frame(tenure="short",
                   spend="high"))

#-#   tenure spend
#-# 1  short  high

str(new)

#-# 'data.frame': 1 obs. of  2 variables:
#-#  $ tenure: Factor w/ 1 level "short": 1
#-#  $ spend : Factor w/ 1 level "high": 1

res <- data.frame(sapply(1:ncol(new),
                         function(x) factor(new[,x],
                                            levels=lev[,x]),
                                            simplify=FALSE))
colnames(res) <- colnames(new)
str(res)

#-# 'data.frame': 1 obs. of  2 variables:
#-#  $ tenure: Factor w/ 3 levels "long","medium",..: 3
#-#  $ spend : Factor w/ 3 levels "high","low","medium": 1

# To wrap up, let's benchmark our NB model on our basetable
#Let's try it on our basetable:

# Read in all data:
source("http://ballings.co/hidden/aCRM/code/chapter2/read_data_sets.R")

#load the AUC package
if (!require("AUC")) {
```

```
  install.packages('AUC',
                   repos="https://cran.rstudio.com/",
                   quiet=TRUE)
  require('AUC')
}

str(BasetableTRAINbig)

#-# 'data.frame': 1415 obs. of  7 variables:
#-#  $ TotalDiscount         : num  129 0 19.3 0 0 ...
#-#  $ TotalPrice            : num  130 951 870 964 962 ...
#-#  $ TotalCredit           : num  0 0 0 -8.5 -6.18 ...
#-#  $ PaymentType_DD        : num  0 0 0 0 0 0 0 0 0 0 ...
#-#  $ PaymentStatus_Not.Paid: num  0 0 0 0 0 0 0 0 0 0 ...
#-#  $ Frequency             : num  1 6 4 4 4 7 4 3 4 5 ...
#-#  $ Recency               : num  243 310 216 141 8 58 55 345 372 319 ...

BasetableTEST$CustomerID <- BasetableTRAINbig$CustomerID <- NULL

NB <- naiveBayes(x=BasetableTRAINbig, y=yTRAINbig)
predNB <- predict(NB,BasetableTEST, type = "raw", threshold = 0.001)[,2]

AUC::auc(roc(predNB,yTEST))

#-# [1] 0.8488317
```

How does this work when we have continuous predictors? Suppose we want to compute the probability that a record belongs to class 1. We compute the mean and standard deviation (and assume a normal disribution) of all the records with $y = 1$. We can also use other distributions. Now that we have the mean and the standard deviation we can plot the normal distribution. For each individual new record we then locate the actual value on the x-axis of that normal distribution and read the corresponding y-value. That y-value corresponds to the likelihood term in Equation 2.7. Multiplying that value with the prior and dividing the result by the normalizing constant will yield the probability that the new record belongs to class 1. Naive bayes is obviously flawed because of the independence assumption. However, it is fast and gives a good baseline for more complex algorihtms.

2.5.4 Logistic regression

Logistic regression is part of the generalized linear model family. Note the difference between a general linear model and a generalized linear model. The former is an ordinary linear (least squares) model with a continuous response variable, and the latter extends the former by generalizing it to situations where the response variable is binary. Consider a situation where y={0,1} and where our fitted equation for instance *i* on the data looks like this:

$$z_i = \beta_0 + \beta_1 x_{i,1} + \beta_2 x_{i,2} + ... + \beta_p x_{i,q} + \varepsilon \tag{2.9}$$

The problem with that equation is that it is unbounded. The linear combination z goes from negative infinity to positive infinity. In logistic regression the solution to that problem is to apply the logistic function, also called sigmoid function, to z. It relates the output of the logistic regression equation (z) to probabilities as follows:

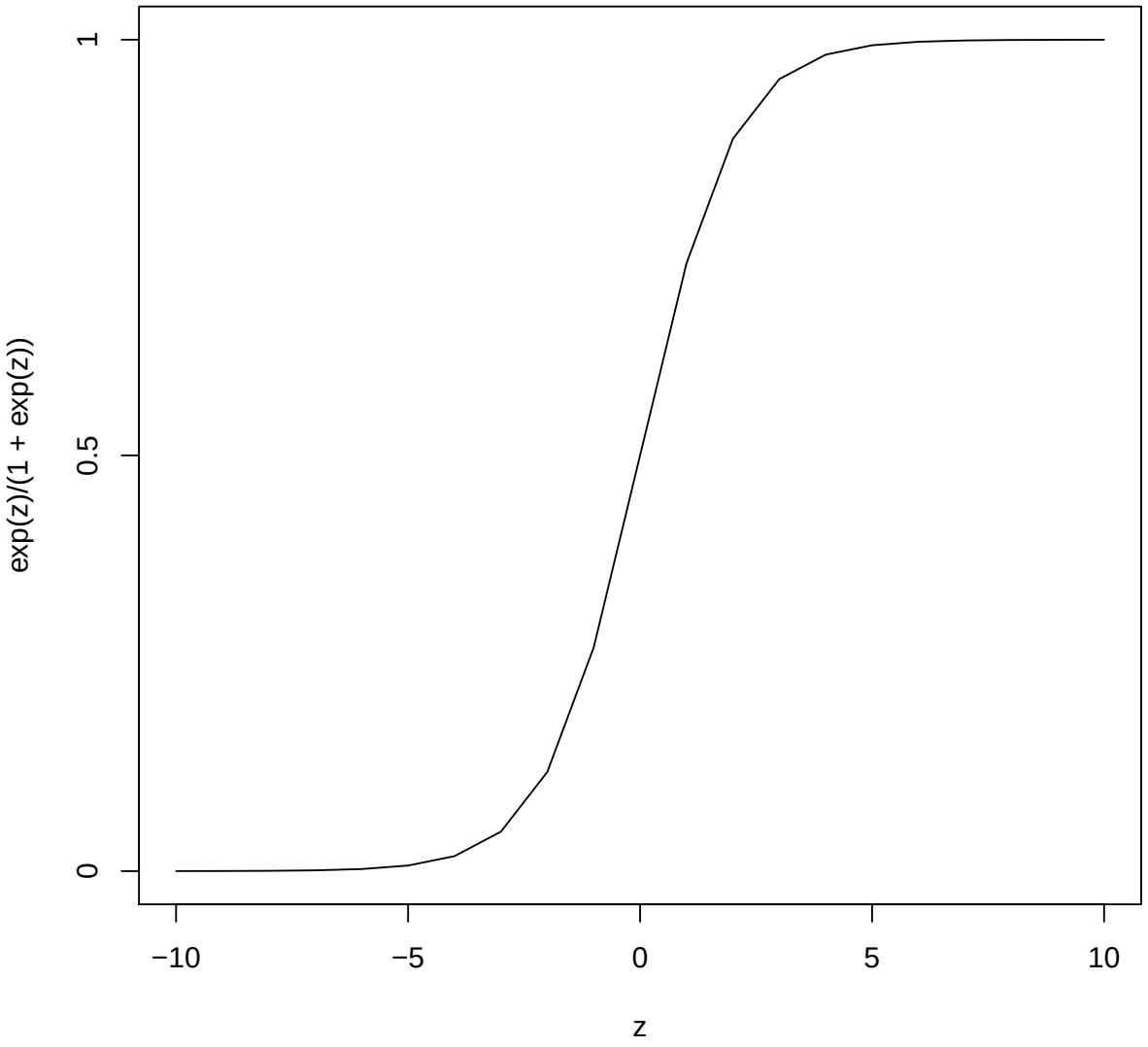
$$P(x_i) = \frac{e^{z_i}}{1 + e^{z_i}} \tag{2.10}$$

$$= \frac{1}{\frac{1}{e^{z_i}} + 1} \tag{2.11}$$

$$= \frac{1}{1 + e^{-z_i}} \tag{2.12}$$

with *e=2.718* Euler's number, $P(x_i)$ short for $P(y_i = 1|x_i)$ (the probability that y=1), and x_i=$\{x_{i,1}, x_{i,2}, \ldots, x_{i,q}\}$ with *i* denoting the instance. The function has the following form and has the property of bounding z by [0,1].

```
z <- -10:10

plot(z,
     exp(z)/(1+exp(z)),
     type="l",
     yaxt="n")
axis(2,at=c(0,0.5,1),labels=c(0,0.5,1))
```



How do we estimate the coefficients in Equation 2.9? We use maximum likelihood, which is short for maximizing the likelihood function. Before we look at the likelihood function we need to talk about Bernoulli trials.

A Bernoulli trial is each repetition of an experiment involving only two outcomes. For example, flipping a coin once, is a Bernoulli trial, and the result is either heads or tails. We are interested in determining the probability of obtaining one of two options. A binomial distribution gives us the probabilities associated with independent, repeated Bernoulli trials. Let's say we are interested in computing the probability of obtaining 4 heads if we toss a coin 10 times. How does the binomial distribution help us with computing this probability? Consider the probability

$$P(4\ heads\ out\ of\ 10\ coin\ flips) = nbr * P(one\ way), \tag{2.13}$$

with nbr the 'number of ways we can get 4 heads out of 10 coin flips' and P(one way) the probability of one possible way the event can occur. For example, one way we can have 4 heads out of 10 tosses is HHHHTTTTTT (with H=heads, T=tails). $P(one\ way)$ in this case is P(H)*P(H)*P(H)*P(H)*P(T)*P(T)*P(T)*P(T)*P(T)*P(T). Since P(H) and P(T) are both 0.5 if we are working with an unbiased coin, $P(one\ way) = 0.5^{10}$. This is true for any other way we can have 4 heads out of 10 coin flips (e.g., HTHTHTHTTT).

In more general terms we would call obtaining a heads a success, and the probability of a given sequence of outcomes where n is the number of trials, k the number of successes, p the probability of success, and $1-p$ the probability of failure, is denoted by:

$$p^k(1-p)^{n-k}. \tag{2.14}$$

For example, p=(1-p)=0.5, n=10, k=4 results in $0.5^4 0.5^6 = 0.0009765625$. In other words the probability of having HHHHTTTTTT (or any other specific sequence) is 0.0009765625 (1/1024). We could now write out all the possible configurations of 4 heads and 6 tails, compute their probability and add all these probabilities. A shortcut is to take advantage of the fact that the probabilities are identical for each possible sequence, so we just need to know the number of possible configurations. This can be computed using the binomial coefficient, $\binom{n}{k} = \frac{n!}{k!(n-k)!}$. If n=10, and k=4, then $\binom{n}{k} = 210$.

```
n <- 10
k <- 4
factorial(n)/(factorial(k)*factorial(n-k))

#-# [1] 210
```

We can now write the complete equation for the binomial distribution:

$$f(p,n,k) = P(X=k) = \binom{n}{k} p^k(1-p)^{n-k} = \frac{n!}{k!(n-k)!} p^k(1-p)^{n-k} \tag{2.15}$$

for k=[0,n]. Our final probability is 0.2050781 (210*0.0009765625). Now, let's get back to our binomial likelihood function. We will apply the Likelihood function to each and every instance in the dataset separately. Therefore $n=1$. Success or failure, k, is defined by our y. As mentioned earlied, we consider $y \in$ {0,1} (e.g., no-churn, churn). We can now rewrite Equation 2.15 as follows (note that $\frac{1!}{y!(1-y)!}$ becomes 1 for both y=0 and y=1):

$$\frac{n!}{k!(n-k)!} p^k(1-p)^{n-k} = \frac{1!}{y!(1-y)!} p^y(1-p)^{1-y} = p^y(1-p)^{1-y}. \tag{2.16}$$

The final step is simply to realize that we need to do this for all instances and multiply the result. We want to multiply the results because we want to optimize on all instances. In other words, we want to estimate a model on all instances, and if we want to determine how likely it

is that two or more events occur, we need to multiply their individual likelihoods. The likelihood function *L* is then defined as follows:

$$L(\beta_0, \beta_1, ..., \beta_q) = \prod_{i=1}^{n} \left(P(x_i)^{y_i} \Big(1 - P(x_i)\Big)^{1-y_i} \right) \tag{2.17}$$

where *n* is the number of instances, and β={β_1,β_2, . . . , β_q}. Maximizing the likelihood is precisely equivalent to maximizing the log likelihood (the logarithm is a monotonic function) and minimizing the negative log likelihood. We use the logarithm because it allows to make the computations easier: products become sums. Without the logarithm we would have to multiply the likelihoods of each instance to compute the overall likelihood. Multiplying a large number of values is prone to numerical overflow (the number cannot be represented anymore by the computer because it becomes very small). The log likelihood *l* is defined as:

$$l(\beta_0, \beta_1, ..., \beta_q) = log\left(\prod_{i=1}^{n} \left(P(x_i)^{y_i} \Big(1 - P(x_i)\Big)^{1-y_i} \right) \right) \tag{2.18}$$

$$= \sum_{i=1}^{n} \left(y_i log\Big(P(x_i)\Big) + (1 - y_i) log\Big(1 - P(x_i)\Big) \right) \tag{2.19}$$

$$= \sum_{i=1}^{n} \left(y_i log\Big(\frac{1}{1 + e^{-z_i}}\Big) + (1 - y_i) log\Big(1 - \frac{1}{1 + e^{-z_i}}\Big) \right) \tag{2.20}$$

The *z*, and hence the set of β coefficients, that yield the maximum *l* are considered the optimal. Let's compute the log likelihood for an instance *i* with a good and a bad prediction. Using Equation 2.19 we find *l*:

```
loglikelihood <- function(y_i,Px_i) y_i * log(Px_i) + (1-y_i)*log(1-Px_i)

# We are maximizing the log likelihood, so higher values are better.

# Suppose we have a good prediction
loglikelihood(y_i= 1,Px_i= 0.9)

#-# [1] -0.1053605

loglikelihood(y_i= 0,Px_i= 0.1)

#-# [1] -0.1053605

#Suppose we have a bad prediction
loglikelihood(y_i= 1,Px_i= 0.1)

#-# [1] -2.302585

loglikelihood(y_i= 0,Px_i= 0.9)

#-# [1] -2.302585

#Clearly the log likelihood of the bad prediction is a lot lower.
```

Overfitting is also in logistic regression a problem. We can cope with overfitting by using stepwise variable selection (forward, backward, bidirectional). In forward selection the variables are added one by one, starting with the variable that improves the model the most, and stay in the model only if they are significant at the moment when they are included in the model. This is computationally expensive because the model needs to be estimated Q times, where Q is the number of variables. In contrast, in backward selection we start with all variables in the model, and remove the variables one by one, starting with the variable that improves the model the most. In forward selection, once a variable is included, it stays in the model, even if it becomes insignificant in subsequent steps. In backward selection, excluded variables stay out of the model even if they were to become significant once there are less variables. Finally, bidirectional selection is a combination of the above, testing at each step for variables to be excluded or included. A better and more efficient way to cope with overfitting is regularization. Regularization keeps all the variables in the model and shrinks the coefficients towards zero. It is also called shrinkage or penalization. We will use the LASSO approach (Tibshirani, 1996), which stands for Least Absolute Shrinkage and Selection Operator. The LASSO log likelihood is defined as follows:

$$l(\beta_0, \beta_1, ..., \beta_q)_{lasso} = -\sum_{i=1}^{n} \left(y_i log\Big(\frac{1}{1+e^{-z_i}}\Big) + (1-y_i) log\Big(1 - \frac{1}{1+e^{-z_i}}\Big)\right) + \lambda \sum_{q=0}^{Q} |\beta_{i,q}| \tag{2.21}$$

$$= -l(\beta_0, \beta_1, ..., \beta_q) + \lambda \sum_{q=0}^{Q} |\beta_{i,q}| \tag{2.22}$$

with Q the number of predictors. The additional term $\lambda \sum_{q=0}^{Q} |\beta_{i,q}|$ is called the L1- norm and represents the sum of the absolute values of all the coefficients. To be clear we will be minimizing $l(\beta_0, \beta_1, ..., \beta_q)_{lasso}$:

$$\underset{\beta_0 \to Q}{\arg\min} \left(-l(\beta_0, \beta_1, ..., \beta_q) + \lambda \sum_{q=0}^{Q} |\beta_{i,q}| \right). \tag{2.23}$$

The λ parameter controls how much the L1-norm impacts regularization and needs to be cross-validated. There is no way to know the optimal value of λ in advance. Higher values of λ will result in smaller coefficients (i.e., more regularization).

In the following code chunk we will see how to run both stepwise and regularized logistic regression, and how to tune the λ parameter. The code can be downloaded from:
`http://ballings.co/hidden/aCRM/code/chapter2/ModelingLR.R`

```
# Read in all data:
source("http://ballings.co/hidden/aCRM/code/chapter2/read_data_sets.R")

#load the AUC package
if (!require("AUC")) {
  install.packages('AUC',
```

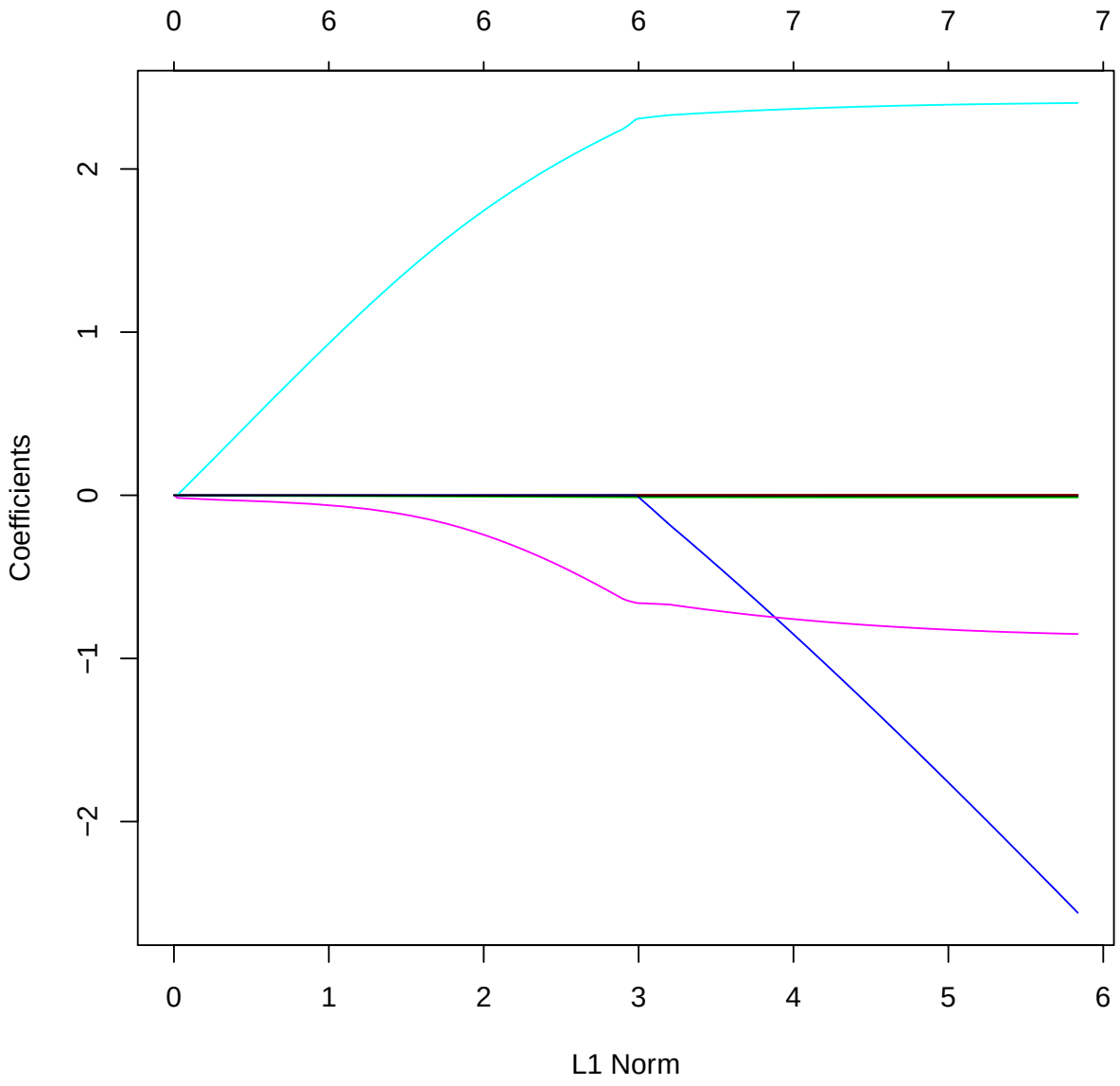
```
                  repos="https://cran.rstudio.com/",
                  quiet=TRUE)
  require('AUC')
}

# Please note that logistic regression is one form of a
# generalized generalized linear model and therefore
# the functions that we will use have the acronym glm
# in their names.


# Option 1: Logistic regression with bi-directional stepwise
# variable selection.
# We will not look at forward and backward selection as
# bi-directional selection is most robust to overfitting.

# As you will see, the glm function will issue a warning.
# We should always investigate what is going on.
# glm.fit: fitted probabilities numerically 0 or 1 occurred
# There were 50 or more warnings (use warnings() to see the first 50)
# There are many possible causes of this warning, such as a perfect predictor
# variable, correlated predictors or the assumption of a linear relationship
# between the independents and the log odds of the dependent.
# In our case it seems to be the assumption of independence of predictors.
# However, the glm function only issues a warning and not an error.
# That is not a big deal for us, since we are only interested in the
# prediction and not in parameter values. We want to determine the
# performance of logistic regression on our data, regardless of
# the assumptions of the algorithm.

(LR <- glm(yTRAINbig ~ .,
           data=BasetableTRAINbig,
           family=binomial("logit")))

#-# Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

#-#
#-# Call:  glm(formula = yTRAINbig ~ ., family = binomial("logit"), data = BasetableTRAINbig)
#-#
#-# Coefficients:
#-#            (Intercept)           TotalDiscount
#-#              3.473e-01               3.007e-03
#-#             TotalPrice             TotalCredit
#-#              5.117e-05              -1.342e-02
#-#         PaymentType_DD  PaymentStatus_Not.Paid
#-#             -1.212e+01               2.550e+00
#-#              Frequency                 Recency
#-#             -8.441e-01              -5.338e-03
#-#
#-# Degrees of Freedom: 1414 Total (i.e. Null);  1407 Residual
#-# Null Deviance:     620.7
#-# Residual Deviance: 469.1  AIC: 485.1
```

```
#Note that '~' means 'regress on' and '~ .' means 'regress on all variables'.

#stepwise variable selection
(LRstep <- step(LR, direction="both", trace = FALSE))

#-# Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
#-# Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
#-# Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
#-# Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
#-# Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
#-# Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
#-# Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
#-# Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
#-# Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
#-# Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
#-# Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
#-# Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
#-# Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
#-# Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
#-# Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
#-# Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
#-# Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
#-# Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
#-# Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
#-# Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
#-# Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
#-# Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

#-#
#-# Call:  glm(formula = yTRAINbig ~ TotalDiscount + TotalCredit + PaymentStatus_Not.Paid +
#-#     Frequency + Recency, family = binomial("logit"), data = BasetableTRAINbig)
#-#
#-# Coefficients:
#-#            (Intercept)           TotalDiscount
#-#               0.344971                0.002968
#-#            TotalCredit  PaymentStatus_Not.Paid
#-#              -0.013397                2.557705
#-#              Frequency                 Recency
#-#              -0.833912               -0.005329
#-#
#-# Degrees of Freedom: 1414 Total (i.e. Null);  1409 Residual
#-# Null Deviance:     620.7
#-# Residual Deviance: 469.3  AIC: 481.3

#Indeed this model contains less variables

#Use the model to make a prediction on test data.
predLRstep <- predict(LRstep,
                      newdata=BasetableTEST,
                      type="response")
```

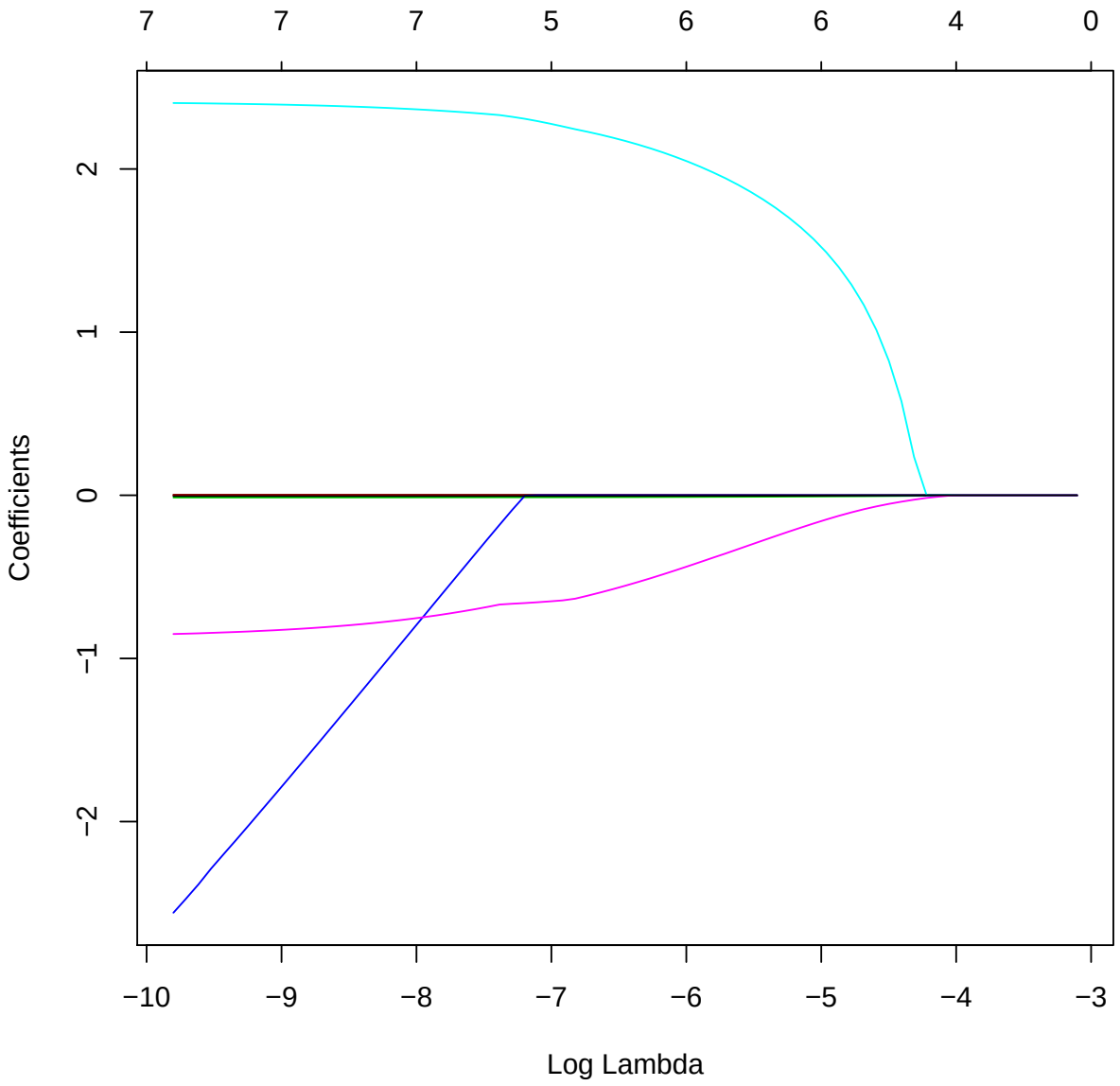
```
#Next we assess the performance of the model
AUC::auc(roc(predLRstep,yTEST))

#-# [1] 0.878078

#Option 2: Regularized Logistic Regression.

#Regularization refers to the introduction of a penalty for complexity
#in order to avoid overfitting. This boils down to keeping all variables
#in the equation but shrinking their coefficients towards 0. Regularization
#is often called shrinkage or lasso (least absolute shrinkage and selection
#operator). More concretely, we set a bound on the sum of the absolute values
#of the coefficients.

#We use the glmnet package:

if (!require("glmnet")) {
  install.packages('glmnet',
                   repos="https://cran.rstudio.com/",
                   quiet=TRUE)
  require('glmnet')
}

# When we load the glmnet package we see this:
# The following object is masked from ‘package:AUC’:
#
#     auc

#This means that there is naming conflict between the AUC package and the
#glmnet package. Therefore we need to append the name of the package with ::
#AUC::auc instead of auc.

(LR <-  glmnet(x=data.matrix(BasetableTRAIN),
               y=yTRAIN,
               family="binomial"))

#-#
#-# Call:  glmnet(x = data.matrix(BasetableTRAIN), y = yTRAIN, family = "binomial")
#-#
#-#       Df       %Dev    Lambda
#-#  [1,]  0 -4.966e-15 4.492e-02
#-#  [2,]  1  1.697e-02 4.093e-02
#-#  [3,]  1  3.018e-02 3.729e-02
#-#  [4,]  1  4.063e-02 3.398e-02
#-#  [5,]  1  4.900e-02 3.096e-02
#-#  [6,]  1  5.575e-02 2.821e-02
#-#  [7,]  1  6.124e-02 2.570e-02
#-#  [8,]  1  6.571e-02 2.342e-02
#-#  [9,]  2  6.959e-02 2.134e-02
#-# [10,]  2  7.318e-02 1.944e-02
#-# [11,]  4  7.820e-02 1.772e-02
```

```
#-# [12,]  4  8.509e-02 1.614e-02
#-# [13,]  5  9.148e-02 1.471e-02
#-# [14,]  6  1.015e-01 1.340e-02
#-# [15,]  6  1.112e-01 1.221e-02
#-# [16,]  6  1.196e-01 1.113e-02
#-# [17,]  6  1.272e-01 1.014e-02
#-# [18,]  6  1.342e-01 9.237e-03
#-# [19,]  6  1.406e-01 8.417e-03
#-# [20,]  6  1.467e-01 7.669e-03
#-# [21,]  6  1.523e-01 6.988e-03
#-# [22,]  6  1.575e-01 6.367e-03
#-# [23,]  6  1.622e-01 5.801e-03
#-# [24,]  6  1.665e-01 5.286e-03
#-# [25,]  6  1.705e-01 4.816e-03
#-# [26,]  6  1.740e-01 4.388e-03
#-# [27,]  6  1.773e-01 3.999e-03
#-# [28,]  6  1.802e-01 3.643e-03
#-# [29,]  6  1.828e-01 3.320e-03
#-# [30,]  6  1.850e-01 3.025e-03
#-# [31,]  6  1.871e-01 2.756e-03
#-# [32,]  6  1.890e-01 2.511e-03
#-# [33,]  6  1.907e-01 2.288e-03
#-# [34,]  6  1.921e-01 2.085e-03
#-# [35,]  6  1.934e-01 1.900e-03
#-# [36,]  6  1.945e-01 1.731e-03
#-# [37,]  6  1.955e-01 1.577e-03
#-# [38,]  6  1.963e-01 1.437e-03
#-# [39,]  6  1.971e-01 1.309e-03
#-# [40,]  6  1.977e-01 1.193e-03
#-# [41,]  6  1.983e-01 1.087e-03
#-# [42,]  5  1.986e-01 9.905e-04
#-# [43,]  5  1.988e-01 9.025e-04
#-# [44,]  5  1.989e-01 8.223e-04
#-# [45,]  6  1.990e-01 7.493e-04
#-# [46,]  6  1.992e-01 6.827e-04
#-# [47,]  7  1.994e-01 6.220e-04
#-# [48,]  7  1.997e-01 5.668e-04
#-# [49,]  7  2.001e-01 5.164e-04
#-# [50,]  7  2.003e-01 4.706e-04
#-# [51,]  7  2.006e-01 4.288e-04
#-# [52,]  7  2.008e-01 3.907e-04
#-# [53,]  7  2.010e-01 3.560e-04
#-# [54,]  7  2.011e-01 3.243e-04
#-# [55,]  7  2.013e-01 2.955e-04
#-# [56,]  7  2.014e-01 2.693e-04
#-# [57,]  7  2.015e-01 2.453e-04
#-# [58,]  7  2.016e-01 2.236e-04
#-# [59,]  7  2.016e-01 2.037e-04
#-# [60,]  7  2.017e-01 1.856e-04
#-# [61,]  7  2.018e-01 1.691e-04
#-# [62,]  7  2.018e-01 1.541e-04
```

```
#-# [63,]  7  2.018e-01 1.404e-04
#-# [64,]  7  2.019e-01 1.279e-04
#-# [65,]  7  2.019e-01 1.166e-04
#-# [66,]  7  2.019e-01 1.062e-04
#-# [67,]  7  2.020e-01 9.677e-05
#-# [68,]  7  2.020e-01 8.817e-05
#-# [69,]  7  2.020e-01 8.034e-05
#-# [70,]  7  2.020e-01 7.320e-05
#-# [71,]  7  2.020e-01 6.670e-05
#-# [72,]  7  2.020e-01 6.077e-05
#-# [73,]  7  2.020e-01 5.538e-05

#Df is the number of variables that are active (i.e., coefficient > 0)
#%Dev is an evaluation metric we are not really interested in.
#Lambda is the degree to which the sum of the absolute values of the
#coefficients is penalized. Higher lambda means that coefficients will
#be smaller.

#For every lambda there is a set of coefficients. The number of active
#variables can also be found by looking at LR

#Let's look at the first and last 2 values of lambda
#For the highest value of lambda we only have an intercept,
#for the second highest value of lambda TotalPrice becomes
#non-zero, ...
coef(LR)[,1:2]

#-# 8 x 2 sparse Matrix of class "dgCMatrix"
#-#                               s0            s1
#-# (Intercept)            -2.92525 -2.7396665744
#-# TotalDiscount           .        .
#-# TotalPrice              .       -0.0002450362
#-# TotalCredit             .        .
#-# PaymentType_DD          .        .
#-# PaymentStatus_Not.Paid  .        .
#-# Frequency               .        .
#-# Recency                 .        .

coef(LR)[,72:73]

#-# 8 x 2 sparse Matrix of class "dgCMatrix"
#-#                                    s71           s72
#-# (Intercept)            -0.1051642036 -0.1035063826
#-# TotalDiscount           0.0030621990  0.0030687503
#-# TotalPrice              0.0006317433  0.0006385955
#-# TotalCredit            -0.0137407155 -0.0137544846
#-# PaymentType_DD         -2.4729864332 -2.5588841878
#-# PaymentStatus_Not.Paid  2.4034331834  2.4041878995
#-# Frequency              -0.8483882960 -0.8503399073
#-# Recency                -0.0047114813 -0.0047144152

#?plot.glmnet
plot(LR)
```

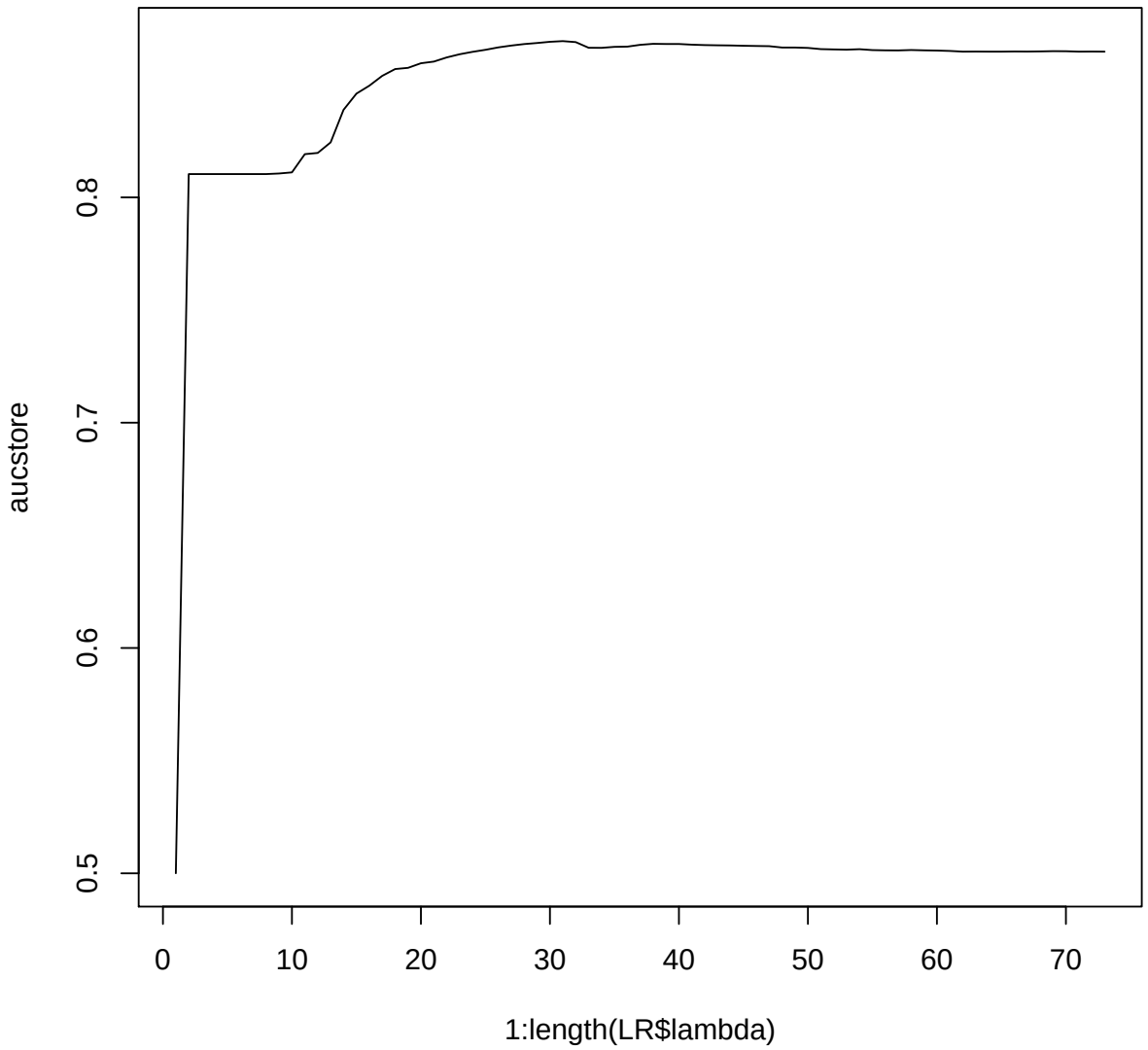



```
#L1 norm= sum of the absolute values of the coefficients
#This shows the obvious: the norm grows when
# the coefficients grow (the norm is the sum
# of the absolute values of the coefficients.

plot(LR,xvar="lambda")
```




```
#Bigger lambda results in more coefficients shrunk to zero.
#min(LR$lambda) max(LR$lambda)


#Cross- validate lambda
#Note that we are not re-estimating the model. We are merely
#redeploying (predict) the model. This makes cross-validation
#very efficient.

aucstore <- numeric()
```

```
for (i in 1:length(LR$lambda) ) {
          predglmnet <- predict(LR,
                                newx=data.matrix(BasetableVAL),
                                type="response",
                                s=LR$lambda[i])
          aucstore[i] <- AUC::auc(roc(as.numeric(predglmnet),yVAL))
}

#Let's determine the optimal lambda value
plot(1:length(LR$lambda),aucstore,type="l")
```



```
(LR.lambda <- LR$lambda[which.max(aucstore)])

#-# [1] 0.002756062

#Now that we know the optimal lambda we can fit the model
#on BasetableTRAINbig.
LR <-  glmnet(x=data.matrix(BasetableTRAINbig),
              y=yTRAINbig,
              family="binomial")

#We then use that model with the optimal lambda.
predLRlas <- as.numeric(predict(LR,
                                newx=data.matrix(BasetableTEST),
                                type="response",
                                s=LR.lambda))


#Finally we assess the performance of the model
AUC::auc(roc(predLRlas,yTEST))

#-# [1] 0.8790811

#Let's unload the glmnet package
detach("package:glmnet", unload=TRUE)
```

2.5.5 Neural networks

Neural networks consist of several layers of nodes. Nodes are also called neurons. Figure 2.8 contains an example of a neural network. There are three layers of nodes: (1) the input nodes are the independent variables (x_1 and x_2), including a constant x_0 to compute the intercepts, (2) the output node is the dependent variable (y), and (3) the hidden nodes are latent variables (x_3 and x_4: weighted sums of the input nodes). There can be multiple hidden layers, and more hidden layers results in higher flexibility of the model. The term 'deep learning' refers to neural networks with many hidden layers. More flexibility is not always a good thing because it makes the model more prone to overfitting. For most problems, one hidden layer is sufficient. The optimal number of hidden nodes needs to be determined by cross-validation.

The lines between nodes are called interlayer connections, and the strength of these connections is denoted by a weight *w*. A special type of connections are skip-layer connections: they do not pass through the hidden layer (w_7, w_8, and b_3). The values b_j are biases (intercepts). The z in the hidden nodes, and the output node, are defined as follows:

$$z = \sum_{j=1}^{p} w_j x_j + b \tag{2.24}$$

with b the intercept, w_j the incoming weights, x_j the incoming nodes, and p the total number of incoming nodes. The function α in Figure 2.8 is called the activation function and is used to avoid

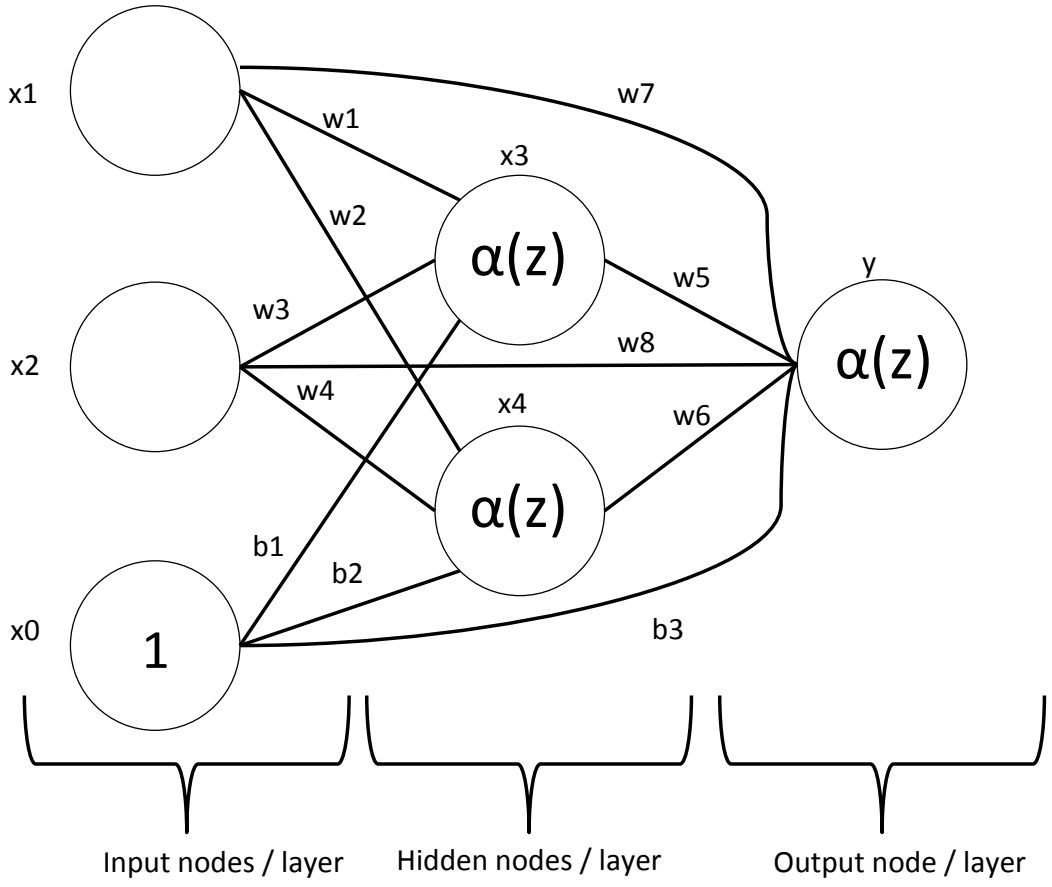


Figure 2.8: Example of a neural network

numerical overflow. This happens when z becomes too large or too small for the computer the store. The logistic activation function is by far the best function to cope with this issue because it bounds z by 0 and 1. It is the same function as used in logistic regression (there it is called the link function):

$$\alpha(z) = \frac{e^z}{1+e^z} \tag{2.25}$$

$$= \frac{1}{\frac{1}{e^z}+1} \tag{2.26}$$

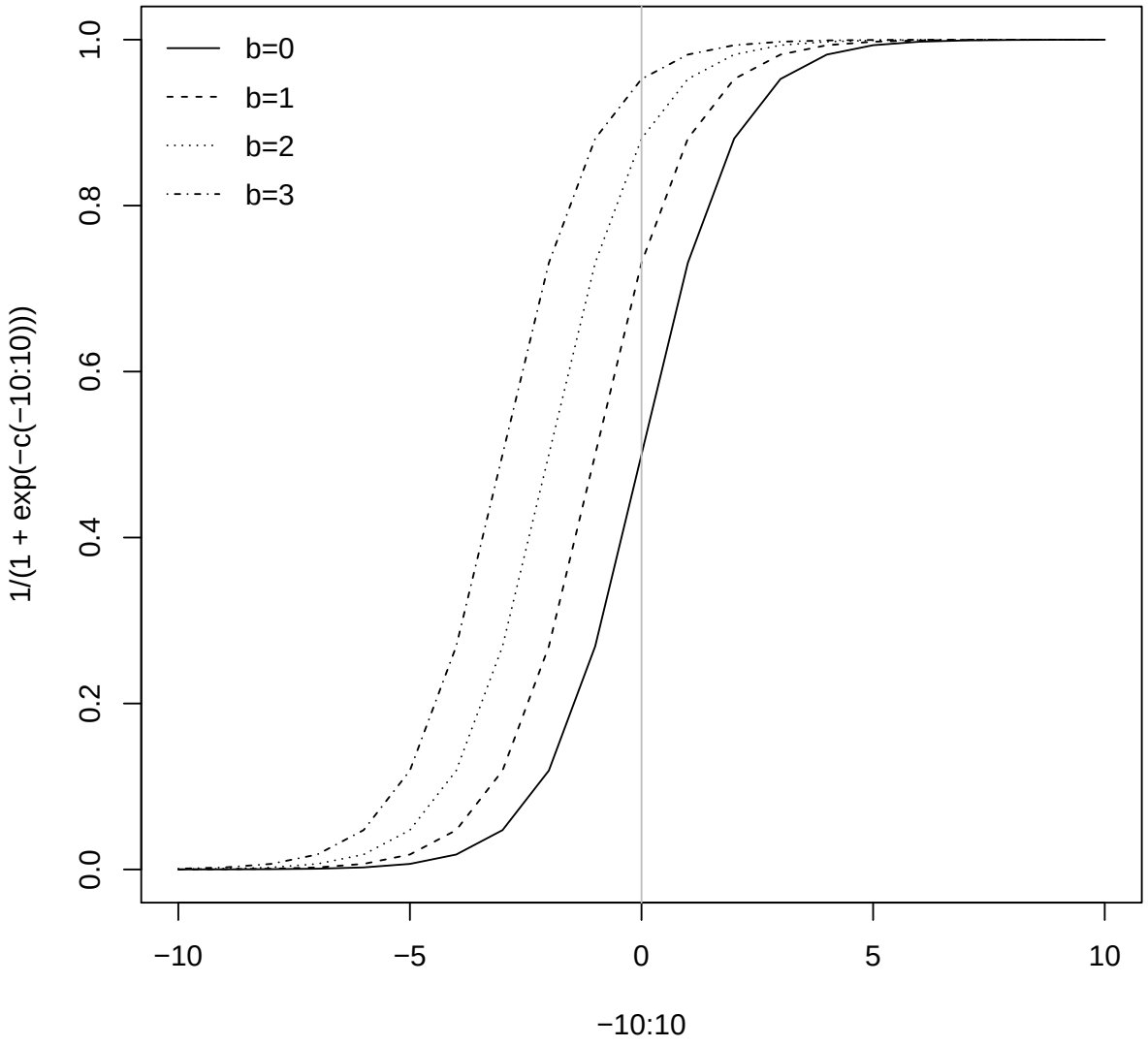
$$= \frac{1}{1+e^{-z}} \tag{2.27}$$

In the case of the output node, $\alpha(z)$ is called $\hat{y}$. Without a hidden layer, and when using the logistic activation function, a neural network is equivalent to a logistic regression. To compute the weights and biases a method called back-propagation is very popular:

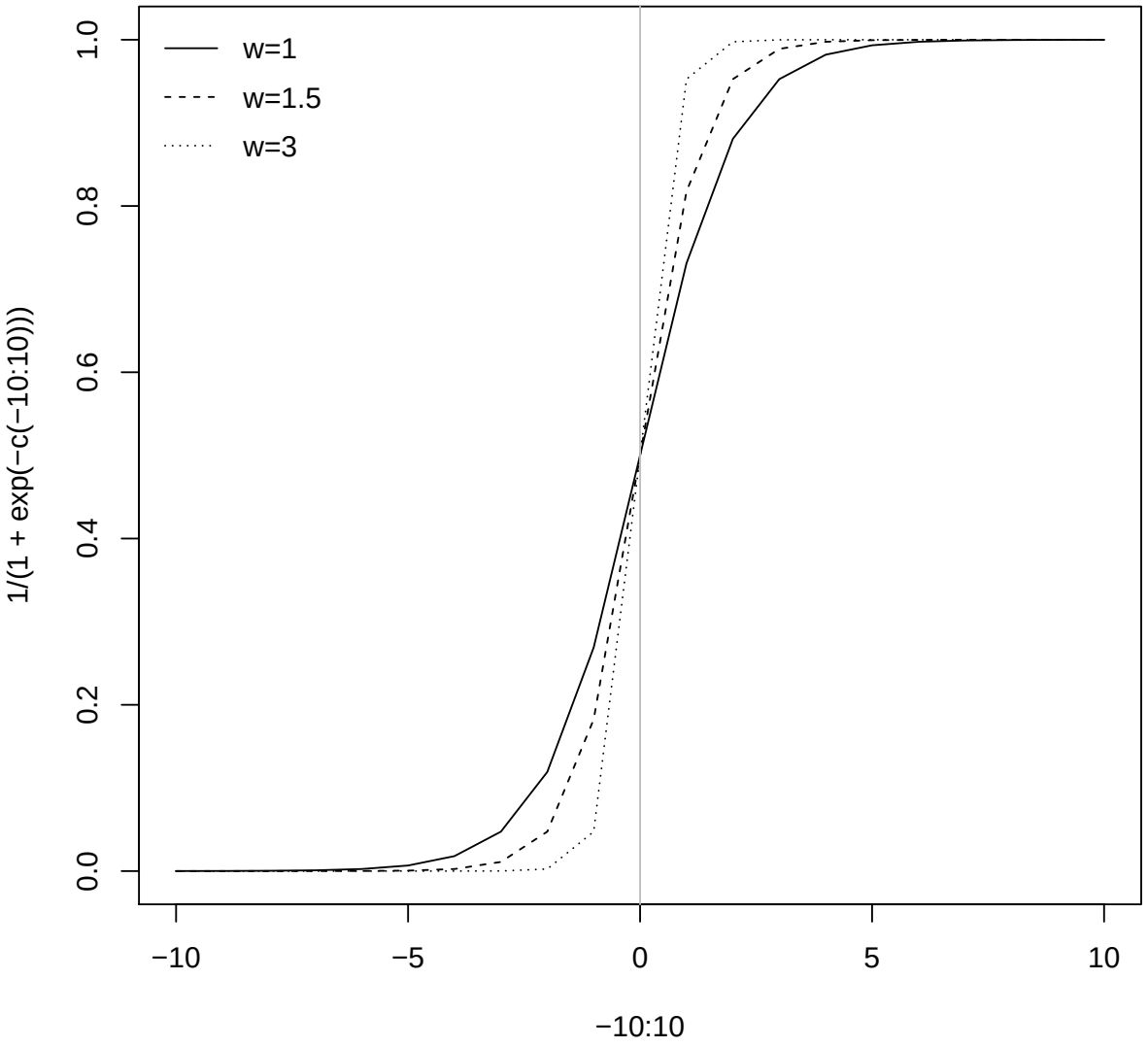
1. Start with random weights
2. Iteratively adjust the weights in the network so as to minimize the cost function (see later) as follows. For each iteration:
   (a) inputs are presented to the network and propagated forward to determine the output
   (b) the predicted output is compared with the observed dependent variable and information about that comparison (how large is the error?) is back-propagated through the network to adjust the weights (large error means large adjustments), and
   (c) this is repeated until the error is low enough

The set of weights and biases will change during the process. Higher values for b shift the logistic curve leftwards and higher values for for w make the curve steeper. The value w measures how sensitive a neuron is to its inputs.

```
#changing b(ias)=intercept: higher values shift the curve
#leftwards. From the perspective of x=0, it looks like the
#curve shifts upwards
plot(-10:10,1/(1+exp(-c(-10:10))),
    ylim=c(0,1),type="l")
lines(-10:10,1/(1+exp(-(c(-10:10)+1))), lty=2)
lines(-10:10,1/(1+exp(-(c(-10:10)+2))), lty=3)
lines(-10:10,1/(1+exp(-(c(-10:10)+3))), lty=4)
legend("topleft", bty="n",
  legend=c("b=0","b=1","b=2","b=3"),
  lty=c(1:4),
  y.intersp=1.4)
abline(v=0,col="grey")
```




```
#changing w: higher values of w make the curve steeper
plot(-10:10,1/(1+exp(-c(-10:10))),
    ylim=c(0,1),type="l")
lines(-10:10,1/(1+exp(-(1.5*c(-10:10)))), lty=2)
lines(-10:10,1/(1+exp(-(3*c(-10:10)))), lty=3)
legend("topleft", bty="n",
  legend=c("w=1","w=1.5","w=3"),
  lty=c(1:3),
  y.intersp=1.4)
abline(v=0,col="grey")
```



The objective fuction that is used in the optimization process is a cost function (it is non-negative, meaning $>= 0$). The goal is to find *w* and *b* for which the function obtains its minimum:

$$\arg\min_{w,b} \left(E\big(y, \hat{y}(x, w, b)\big) + \frac{\lambda}{2n} \sum_{j=1}^{P} w_j^2 \right) \tag{2.28}$$

where $\hat{y}$ is the predicted output, E is the entropy (always non-negative; see below), λ is the shrinkage parameter, $\frac{\lambda}{2n} \sum_{j=1}^{P} w_j^2$ is the weight decay also called regularization term or L2-norm, n is the number of instances in the training set, and p is the total number of weights in the entire network. The regularization term does not include the bias because bias does not make a

neuron sensitive to its inputs, while a weight does. Therefore we don't need to worry about large biases enabling our network to learn noise and overfit. Higher values of λ result in smaller weights (good values for λ=[0.01-0.1] for entropy). Finally, we see that the regularization term is divided by *2n*. This means that greater training sets receive less regularization. This is logical in that bigger training sets reduces the variable-to-instance ratio, and therefore reduces the likelihood of overfitting.

In logistic regression we used the L1-norm (also called lasso), and here we use the L2-norm (also called ridge term). Both work well but have somewhat different characteristics. In L1 weights shrink by a constant amount, whereas in L2 the weights shrink by an amount proportional to the weights. When *w* are large, L1 shrinks the weights a lot less than L2. When *w* are small, L1 shrinks weights a lot more. For example, if an initial weight is small, 0.1, then L2 will grow by only 0.01 while L1 grows by 0.1. If the initial weight is 3, then L1 will grow by only 3, while L2 will grow by 9. Therefore, L1 is said to be sparser: it concentrates all weight in a small number of high importance connections, while the smaller weights quickly arrive at 0.

Next, we define entropy, *E*, as follows:

$$E = -\frac{1}{n}\sum_{i=1}^{n}\left(y\ ln(\hat{y}) + (1-y)\ ln(1-\hat{y})\right) \tag{2.29}$$

where *n* is the number of training instances. Let's try to understand that equation by computing its value when we have a good prediction, and when we have a bad prediction.

```
# The equation for only one prediction
# simplifies to:
E <- function(y,yhat) -(y*log(yhat) + (1-y)*log(1-yhat))

#in case of a good prediction
E(y=1,yhat=0.9)

#-# [1] 0.1053605

E(y=0,yhat=0.1)

#-# [1] 0.1053605

#in case of a bad prediction
E(y=1,yhat=0.1)

#-# [1] 2.302585

E(y=0,yhat=0.9)

#-# [1] 2.302585
```

The cost of a bad prediction is a lot higher and since a smaller objective function is better (we are minimizing it; see Equation 2.28) this is exactly what we want.

Next we will look at how to implement a neural network. We will use one hidden layer, and cross- validate the number of hidden nodes. In general, more hidden layers results in more flexibility. Good values for the number of nodes are [1,100]. The following code chunk can be accessed at: http://ballings.co/hidden/aCRM/code/chapter2/ModelingNN.R.

```
# Read in all data:
source("http://ballings.co/hidden/aCRM/code/chapter2/read_data_sets.R")

#load the AUC package
if (!require("AUC")) {
  install.packages('AUC',
                   repos="https://cran.rstudio.com/",
                   quiet=TRUE)
  require('AUC')
}

#load the AUC package
if (!require("nnet")) {
  install.packages('nnet',
                   repos="https://cran.rstudio.com/",
                   quiet=TRUE)
  require('nnet')
}

#-# Loading required package: nnet

#Run an external function to facilitate tuning
source("http://ballings.co/hidden/aCRM/code/chapter2/tuneMember.R")
#first we need to scale the data to range [0,1] avoid numerical problems
BasetableTRAINnumID <- sapply(BasetableTRAIN, is.numeric)
BasetableTRAINnum <- BasetableTRAIN[, BasetableTRAINnumID]

minima <- sapply(BasetableTRAINnum,min)
scaling <- sapply(BasetableTRAINnum,max)-minima
#?scale
#center is subtracted from each column. Because we use the minima this sets the minimum to zero.
#scale: each column is divided by scale. Because we use the range this sets the maximum to one.
BasetableTRAINscaled <- data.frame(base::scale(BasetableTRAINnum,
                                               center=minima,
                                               scale=scaling),
                                   BasetableTRAIN[,!BasetableTRAINnumID])
colnames(BasetableTRAINscaled) <-  c(colnames(BasetableTRAIN)[BasetableTRAINnumID],
                                     colnames(BasetableTRAIN)[!BasetableTRAINnumID])
#check
sapply(BasetableTRAINscaled,range)

#-#      TotalDiscount TotalPrice TotalCredit PaymentType_DD
#-# [1,]             0          0           0              0
#-# [2,]             1          1           1              1
#-#      PaymentStatus_Not.Paid Frequency Recency
#-# [1,]                      0         0       0
#-# [2,]                      1         1       1
```

```
NN.rang <- 0.5 #the range of the initial random weights parameter
NN.maxit <- 10000 #set high in order not to run into early stopping
NN.size <- c(5,10,20) #number of units in the hidden layer
NN.decay <- c(0,0.001,0.01,0.1) #weight decay.
                                #Same as lambda in regularized LR. Controls overfitting

call <- call("nnet",
             formula = yTRAIN ~ .,
             data=BasetableTRAINscaled,
             rang=NN.rang, maxit=NN.maxit,
             trace=FALSE, MaxNWts= Inf)
tuning <- list(size=NN.size, decay=NN.decay)

#tune nnet
#scale validation data
BasetableVALIDATEnum <- BasetableVAL[, BasetableTRAINnumID]
BasetableVALIDATEscaled <- data.frame(base::scale(BasetableVALIDATEnum,
                                                  center=minima,
                                                  scale=scaling),
                                      BasetableVAL[,!BasetableTRAINnumID])
colnames(BasetableVALIDATEscaled) <- colnames(BasetableTRAINscaled)

(result <- tuneMember(call=call,
                      tuning=tuning,
                      xtest=BasetableVALIDATEscaled,
                      ytest=yVAL,
                      predicttype="raw"))

#-#   size decay       auc
#-# 7    5  0.01 0.9297805

#Create final model

BasetableTRAINbignum <- BasetableTRAINbig[, BasetableTRAINnumID]
BasetableTRAINbigscaled <- data.frame(base::scale(BasetableTRAINbignum,
                                                  center=minima,
                                                  scale=scaling),
                                      BasetableTRAINbig[,!BasetableTRAINnumID])
colnames(BasetableTRAINbigscaled) <-  c(colnames(BasetableTRAINbig)[BasetableTRAINnumID],
                                        colnames(BasetableTRAINbig)[!BasetableTRAINnumID])

NN <- nnet(yTRAINbig ~ .,
           BasetableTRAINbigscaled,
           size = result$size,
           rang = NN.rang,
           decay = result$decay,
           maxit = NN.maxit,
           trace=TRUE,
           MaxNWts= Inf)

#-# # weights:  46
```

```
#-# initial  value 1055.686153
#-# iter  10 value 315.415530
#-# iter  20 value 267.402952
#-# iter  30 value 219.864479
#-# iter  40 value 204.822967
#-# iter  50 value 195.534731
#-# iter  60 value 189.568031
#-# iter  70 value 186.985475
#-# iter  80 value 184.359689
#-# iter  90 value 180.420498
#-# iter 100 value 177.418567
#-# iter 110 value 175.139233
#-# iter 120 value 173.261549
#-# iter 130 value 172.083854
#-# iter 140 value 171.916322
#-# iter 150 value 171.870828
#-# iter 160 value 171.353412
#-# iter 170 value 171.242193
#-# iter 180 value 171.238086
#-# iter 190 value 171.237842
#-# final  value 171.237835
#-# converged

#predict on test
BasetableTESTbignum <- BasetableTEST[, BasetableTRAINnumID]
BasetableTESTscaled <- data.frame(base::scale(BasetableTESTbignum,
                                              center=minima,
                                              scale=scaling),
                                  BasetableTESTbignum[,!BasetableTRAINnumID])
colnames(BasetableTESTscaled) <-  c(colnames(BasetableTESTbignum)[BasetableTRAINnumID],
                                    colnames(BasetableTESTbignum)[!BasetableTRAINnumID])

predNN <- as.numeric(predict(NN,BasetableTESTscaled,type="raw"))

auc(roc(predNN,yTEST))

#-# [1] 0.9519314
```

2.5.6 K-nearest neighbors

K-nearest neighbors is a simple but powerful algorithm. It is the only algorithm in which all computations are done in the prediction phase (as opposed to in the estimation phase). The algorithm works as follows. Consider a training dataset and new dataset with two predictors x_1 and x_2. The training set has a dependent variable $y \in \{0, 1\}$. For each new instance *i* of the new dataset:

- compute the Euclidean Distance, or dissimilarity with all training instances:

$$dissimilarity = \sqrt{(x_{1,new} - x_{1,train})^2 + (x_{2,new} - x_{2,train})^2}, \tag{2.30}$$

- take the *k* least dissimilar training instances (i.e., the nearest neighbors),
- take y of those nearest neighbors, and
- compute the proportion of 1s (this will be the predicted score).

The calculation of the dissimilarity is based on the Pythagorean theorem: $c^2 = a^2 + b^2$, $c = \sqrt{a^2 + b^2}$ and is displayed in Figure 2.9

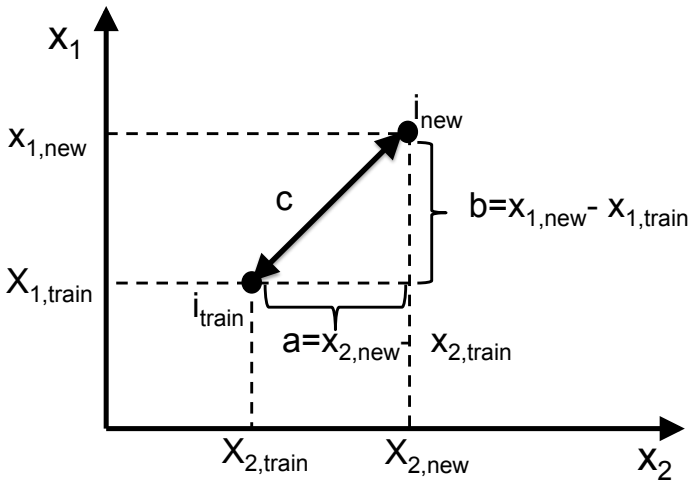


Figure 2.9: Pythagorean theorem

The algorithm is very sensitive to the parameter *k*. Smaller values for *k* are prone to overfitting (learning noise), whereas bigger values are prone to underfitting (not finding the pattern). If *k* is too large, then boundaries will be oversimplified. If $k = N$, with *N* the number of training instances, then we always predict the majority class. In other words, the prediction for all new instances will equal the proportion of 1s in the entire training set.

Consider an example in Figure 2.10. The dots and crosses are training instances. Assume a dot equals 0, and a cross equals 1. The triangle is a new instance and we want to predict its score. When $k = 1$ (represented by the inner circle) a cross will be the triangle's nearest neighbor. Therefore the predicted score will be 1. When $k = 10$ (represented by the outer circle) 9 out of 10 nearest neighbors are dots, and only 1 out of 9 is a cross. Therefore the predicted score for the triangle will be 0.1. Clearly the triangle should be classified as a dot and not as a cross, and therefore $k = 10$ is the better choice here. In practice we tune *k* to make sure we have the optimal value.

The following code is available from: `http://ballings.co/hidden/aCRM/code/chapter2/ModelingKNN.R`

```
# First, read in all data:
source("http://ballings.co/hidden/aCRM/code/chapter2/read_data_sets.R")
```

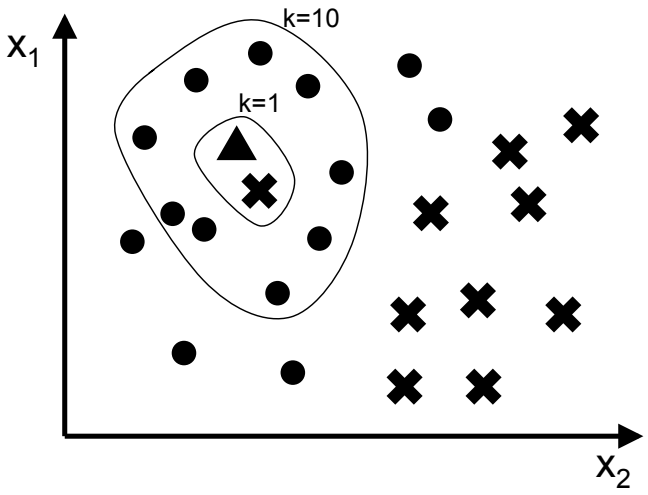
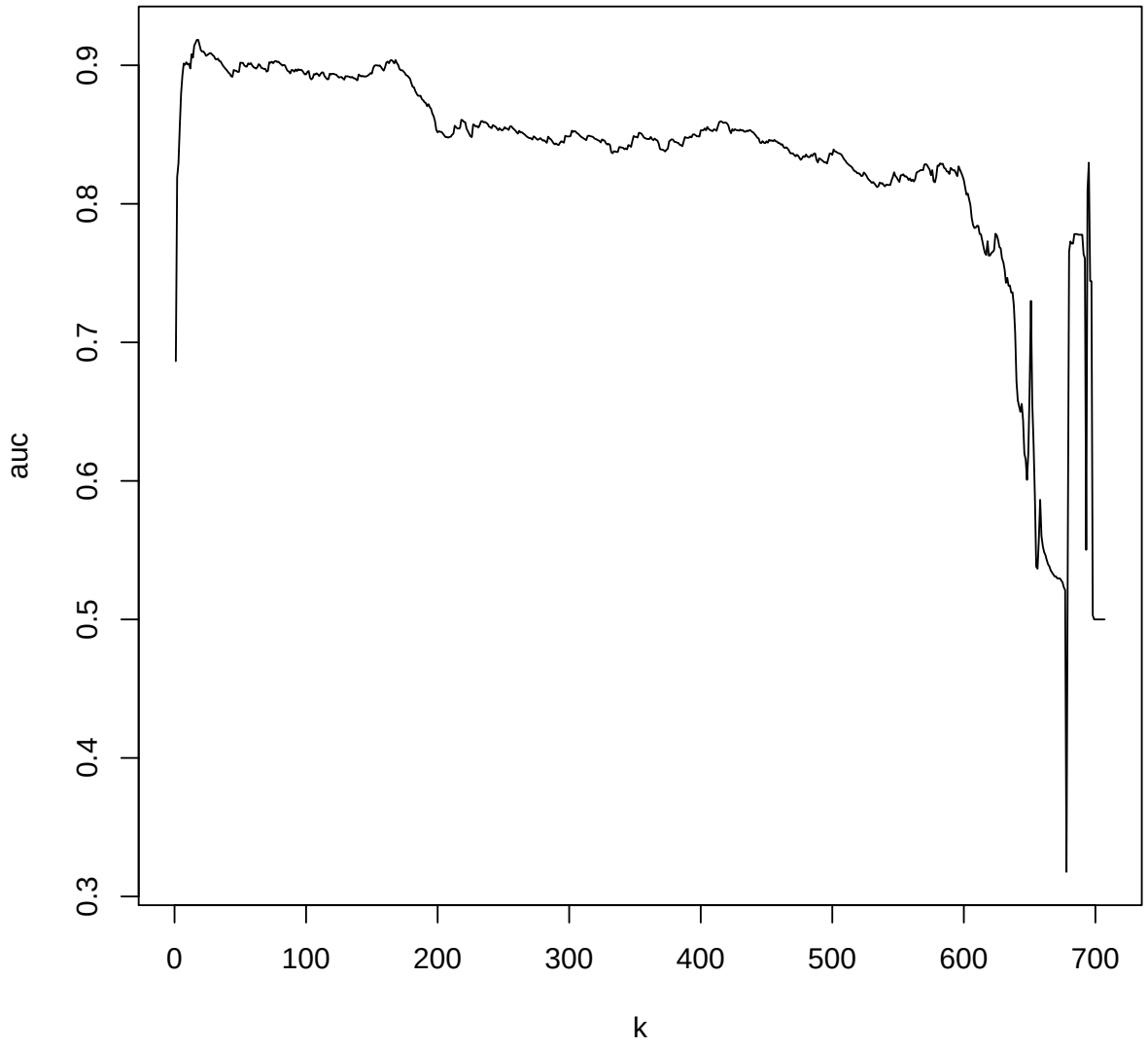


Figure 2.10: The k parameter

```
if (!require("AUC")) {
  install.packages('AUC',
                   repos="https://cran.rstudio.com/",
                   quiet=TRUE)
  require('AUC')
}

#Fast Nearest Neighbor Search Algorithms and Applications
if (!require("FNN")) {
  install.packages('FNN',
                   repos="https://cran.rstudio.com/",
                   quiet=TRUE)
  require('FNN')
}

#-# Loading required package: FNN

# The main function we will be using is the knnx.index function.
# It finds the k-nearest neighbors.The knnx function requires
# all indicators to be numeric so we first convert our data.
# In this case they are already numeric, but this is an example
# of how to do it:
trainKNN <- data.frame(sapply(BasetableTRAIN, function(x) as.numeric(as.character(x))))
trainKNNbig <- data.frame(sapply(BasetableTRAINbig, function(x) as.numeric(as.character(x))))
valKNN <- data.frame(sapply(BasetableVAL, function(x) as.numeric(as.character(x))))
testKNN <- data.frame(sapply(BasetableTEST, function(x) as.numeric(as.character(x))))

# The distance function (e.g., Euclidean distance) is sensitive
# to the scale of the variables. For example, if we have the
# variable 'frequency' measured as number of shop visits, and
# monetary value in dollars the latter will have a higher
```

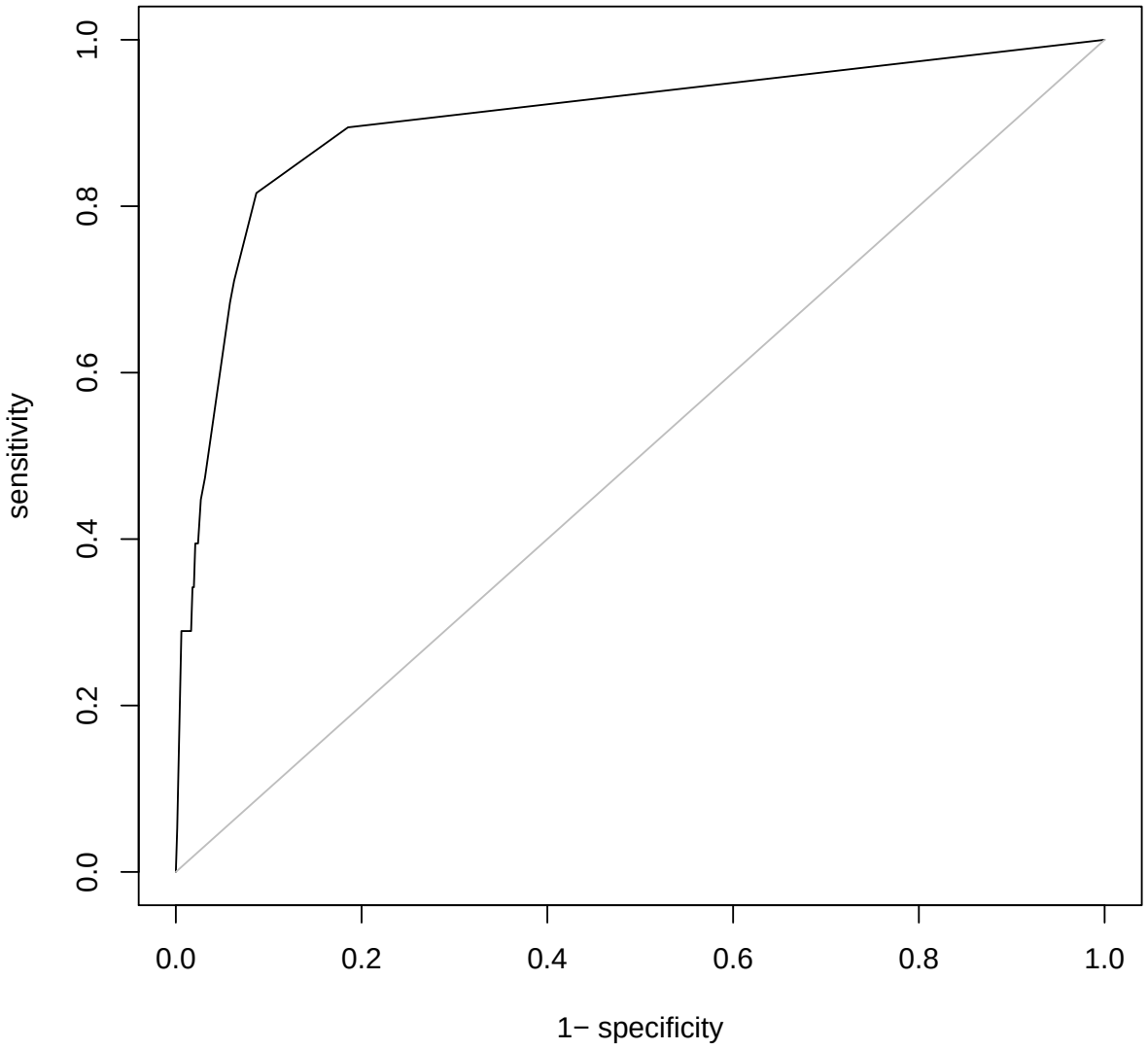
```r
# influence on the distance measured. Therefore we need to
# standardize the variables first.

stdev <- sapply(trainKNN,sd)
means <- sapply(trainKNN,mean)

trainKNNbig <- data.frame(t((t(trainKNNbig)-means)/stdev))
trainKNN <- data.frame(t((t(trainKNN)-means)/stdev))
valKNN <- data.frame(t((t(valKNN)-means)/stdev))
testKNN <- data.frame(t((t(testKNN)-means)/stdev))

#note that all computations take place in the prediction phase

#example for 10 nearest neighbors:
k <- 10
#retrieve the indicators of the k nearest neighbors of the query data
indicatorsKNN <- as.integer(knnx.index(data=trainKNNbig,
                                       query=testKNN,
                                       k=k))
#retrieve the actual y from the training set
predKNN <- as.integer(as.character(yTRAINbig[indicatorsKNN]))
#if k > 1 then we take the proportion of 1s
predKNN <- rowMeans(data.frame(matrix(data=predKNN,
                                      ncol=k,
                                      nrow=nrow(testKNN))))

AUC::auc(roc(predKNN,yTEST))

#-# [1] 0.8813233

#if we want to tune:
#tuning comes down to evaluating which value for k is best
auc <- numeric()
for (k in 1:nrow(trainKNN)) {
  #retrieve the indicators of the k nearest neighbors of the query data
  indicatorsKNN <- as.integer(knnx.index(data=trainKNN,
                                         query=valKNN,
                                         k=k))
  #retrieve the actual y from the training set
  predKNN <- as.integer(as.character(yTRAIN[indicatorsKNN]))
  #if k > 1 then we take the proportion of 1s
  predKNN <- rowMeans(data.frame(matrix(data=predKNN,
                                        ncol=k,
                                        nrow=nrow(valKNN))))

  #COMPUTE AUC
  auc[k] <- AUC::auc(roc(predKNN,yVAL))

  #Print progress to the screen
  if((k %% max(floor(nrow(trainKNN)/10),1))==0)
    cat(round((k/nrow(trainKNN))*100),"%\n")
}
```

```
#-# 10 %
#-# 20 %
#-# 30 %
#-# 40 %
#-# 50 %
#-# 59 %
#-# 69 %
#-# 79 %
#-# 89 %
#-# 99 %

plot(1:nrow(trainKNN),auc,type="l", xlab="k")
```




```
#very low values of k result in a very flexible classifier: low bias, high variance
#very high values of k result a very inflexbile classifier: high bias, low variance
#when k equals the the number of training instances then all
#response values are selected once per new (i.e., validation) data point.
#Then all values of predKNN will have mean(as.integer(as.character(yTRAIN))).

#the next step would be to train again on trainKNNbig using
#the best value of k and predict on testKNN


(k <- which.max(auc))
```

```
#-# [1] 18

#retrieve the indicators of the k nearest neighbors of the query data
indicatorsKNN <- as.integer(knnx.index(data=trainKNNbig,
                                       query=testKNN,
                                       k=k))
#retrieve the actual y from the training set
predKNNoptimal <-
  as.integer(as.character(yTRAINbig[indicatorsKNN]))
#if k > 1 then we take the proportion of 1s
predKNNoptimal <-
  rowMeans(data.frame(matrix(data=predKNNoptimal,
                             ncol=k,
                             nrow=nrow(testKNN))))

AUC::auc(roc(predKNNoptimal,yTEST))

#-# [1] 0.9029777

plot(roc(predKNNoptimal,yTEST))
```



2.5.7 Decision trees

Consider the nonlinear classification problem in Figure 2.11. LOR stands for 'length of relationship' and MON stands for 'monetary value'. The stars denote 1s (e.g., churners) and the dots denote 0s (e.g., stayers). The goal is to separate the stars from the dots by making binary partitions of the data using a series of rules. This is called Binary Recursive Partitioning (BRP) (Breiman et al., 1984) and is displayed in Figure 2.12. The root node contains all the data (all the dots and stars in Figure 2.11). Then, a specific value of a specific variable is chosen to make a split. For example, the root node is split based on the following cutpoint: value 5 of the MON variable. All

the instances that are MON $<$ 5 go to the left node, and all the instances that are MON $>=$ 5 go to the right node. Further splits can then be made. For example, we further partition all the instances that have MON $>=$ 5 using the following cutoff: value 5 of variable LOR.

How are the splits chosen? The optimization backend is exhaustive meaning that all possible splits of all variables are tried, and the best split is chosen. For example, suppose that MON has the following unique values {1,3,5,10} and LOR has {2,4,5,6,10}. The tree would start trying to split the root node using MON(1), MON(3), ..., LOR(5), LOR(6). It would store how good each split is, and then select the best split. At each node a binary partitioning of the data is created using the best split. The algorithm proceeds recursively by, within each parent partition, again evaluating all possible splits, and creating two child partitions based on the best split.

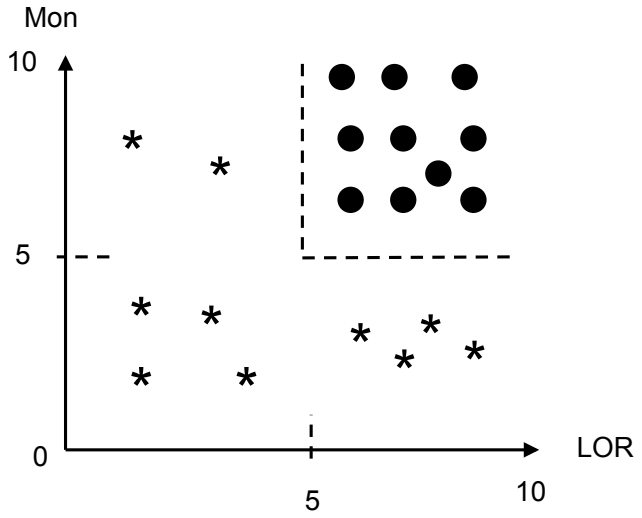


Figure 2.11: An example of a two dimensional space

The best split is defined as the split *s* that maximizes the decrease of impurity of the parent node τ (Berk, 2008, p116), with impurity measured by the Gini index $p(1-p)$:

$$\Delta(s,\tau) = p_\tau(1-p_\tau) - \left(\frac{|\tau_L|}{|\tau|} p_{\tau_L}(1-p_{\tau_L}) + \frac{|\tau_R|}{|\tau|} p_{\tau_R}(1-p_{\tau_R}) \right) \tag{2.31}$$

where p is short for $p(y=1)$ with $y \in \{0,1\}$, and τ, τ_L, τ_R respectively all the cases in the parent node, left child node and right child node. We denote cardinality (i.e., number of instances) by $|\cdot|$. The Gini index is a concave function, having minima at $p=0$ and $p=1$ and a maximum at $p=0.5$. At each decision node, the predictor that yields the purest division of data (i.e., the highest decrease of impurity, $\Delta(s,\tau)$) is selected and the algorithm stops when, for example, a minimum partition size, a specific impurity or a number of partitions is obtained.

Once a tree is built, it can be deployed to predict new samples (with unknown responses) by putting them through the tree. The proportion of 1s in the y variable of the instances in a node that a new sample falls into then becomes the predicted response for the new sample. Recursive partitioning can be seen as a special case of stepwise regression (Berk, 2008, p121). Consider

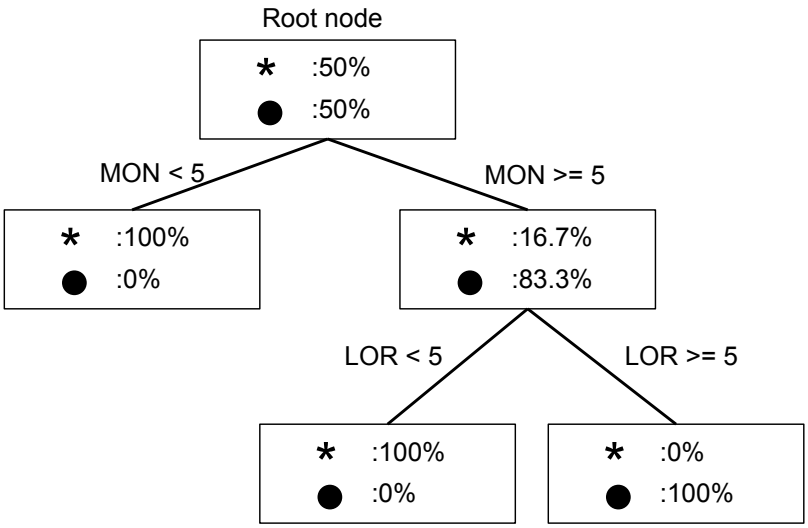


Figure 2.12: Tree built on Figure 2.11

the following example of a tree with three terminal nodes. The root node uses variable x with split value c_1 to partition into terminal node τ_1 to the left and into an internal node to the right. The internal node then uses variable z with split value c_2 to partition into terminal node τ_2 to the left and terminal node τ_3 to the right. The proportions of 1s of the three terminal nodes are $Prop(y_{\tau_1})$, $Prop(y_{\tau_2})$ and $Prop(y_{\tau_3})$. The response of a new observation is then predicted using the following additive model:

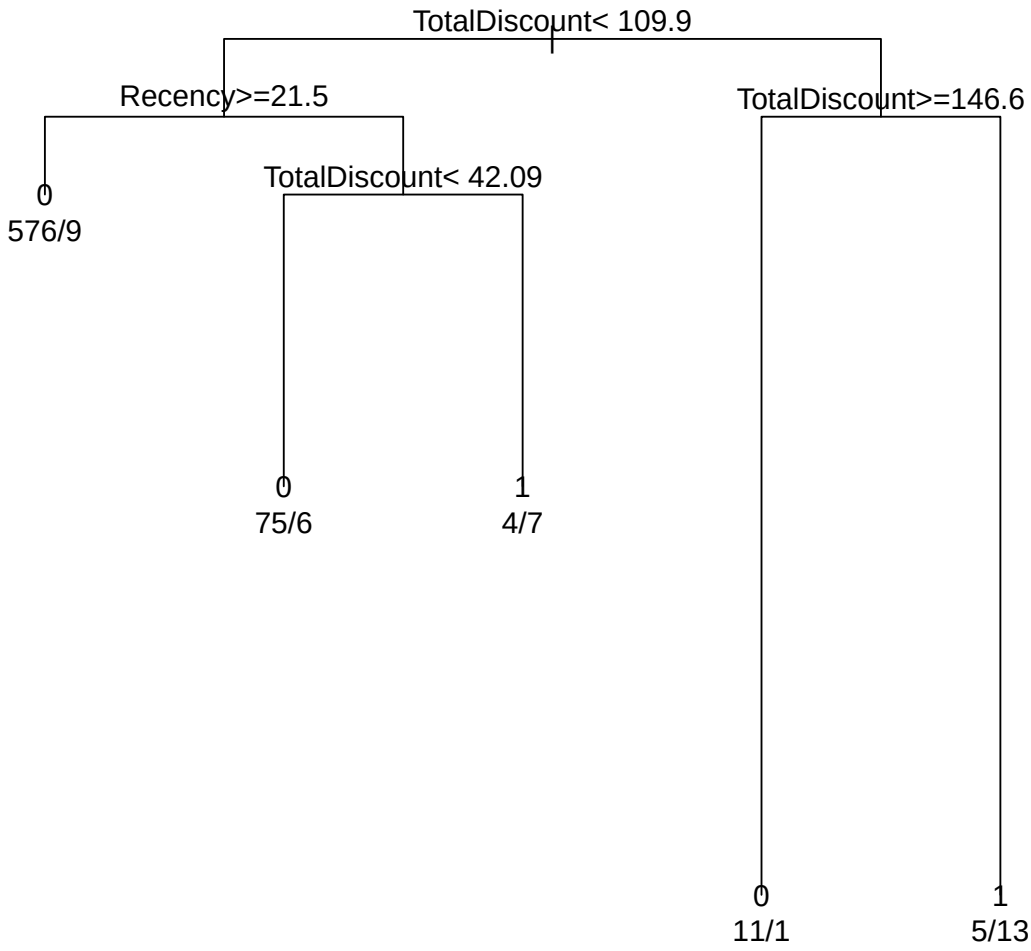
$$f(x,z) = Prop(y_{\tau_1})[I(x <= c_1)] + Prop(y_{\tau_2})[I(x > c_1 \wedge z <= c_2)] + Prop(y_{\tau_3})[I(x > c_1 \wedge z > c_2)] \quad (2.32)$$

where terminal nodes are represented with indicator variables, which are functions of one or more variables.

All code in this section is available at:
`http://ballings.co/hidden/aCRM/code/chapter2/ModelingDT.R`

```
#Download the data
rm(list=ls())
source("http://ballings.co/hidden/aCRM/code/chapter2/read_data_sets.R")

#check what is in our environment
ls()

#-# [1] "BasetableTEST"     "BasetableTRAIN"
#-# [3] "BasetableTRAINbig" "BasetableVAL"
#-# [5] "classes"           "yTEST"
#-# [7] "yTRAIN"            "yTRAINbig"
#-# [9] "yVAL"
```

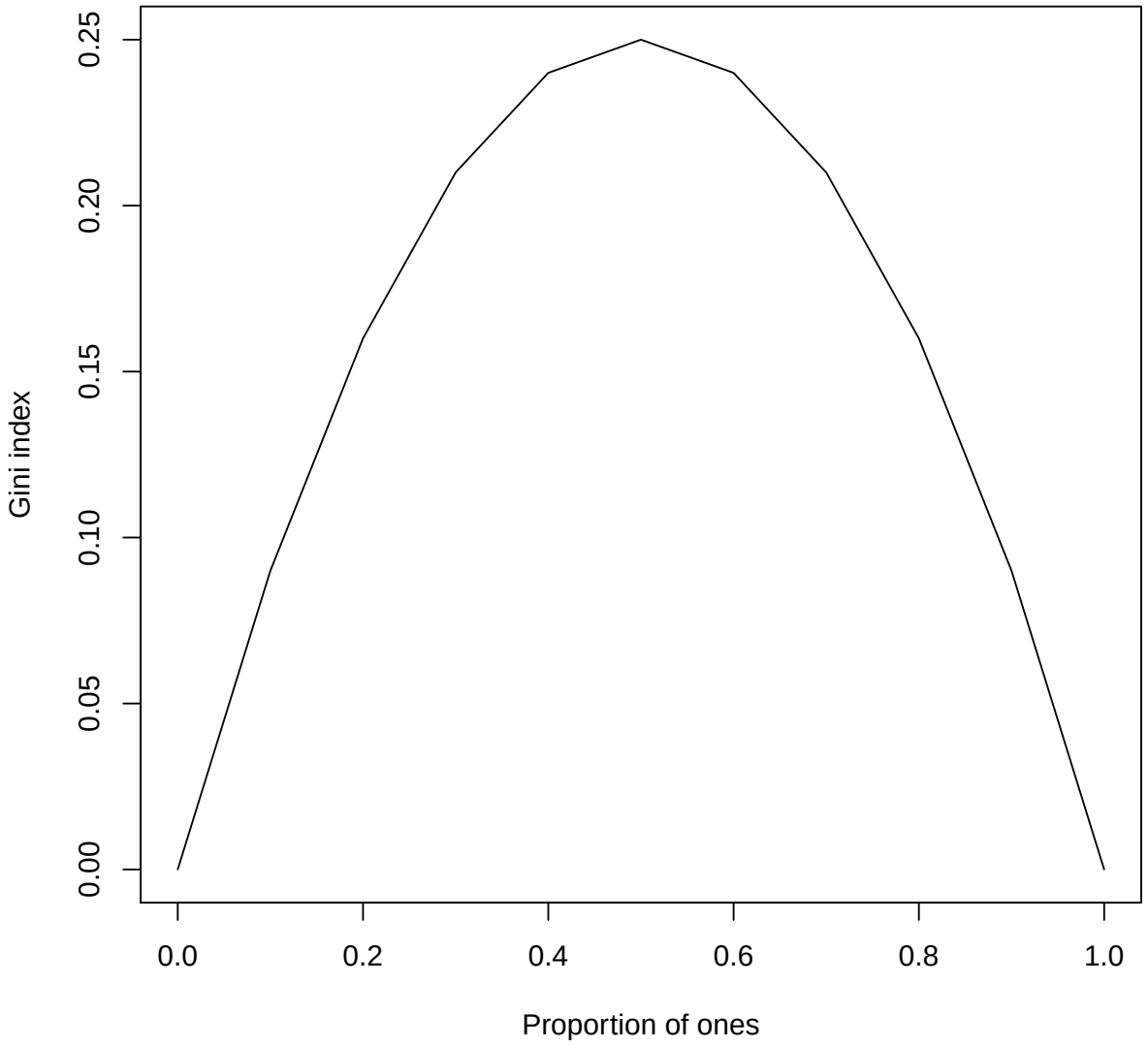
```r
#load the package AUC to evaluate model performance
if(require('AUC')==FALSE)  {
  install.packages('AUC',
                   repos="https://cran.rstudio.com/",
                   quiet=TRUE)
  require('AUC')
}

#load the package rpart to create decision  trees
if(require('rpart')==FALSE)  {
  install.packages('rpart',
                   repos="https://cran.rstudio.com/",
                   quiet=TRUE)
  require('rpart')
}

#-# Loading required package: rpart

#always first look at the main manual page
?rpart
#estimate a tree model
(tree <- rpart(yTRAIN ~ ., BasetableTRAIN))

#-# n= 707
#-#
#-# node), split, n, loss, yval, (yprob)
#-#       * denotes terminal node
#-#
#-#  1) root 707 36 0 (0.94908062 0.05091938)
#-#    2) TotalDiscount< 109.9136 677 22 0 (0.96750369 0.03249631)
#-#      4) Recency>=21.5 585  9 0 (0.98461538 0.01538462) *
#-#      5) Recency< 21.5 92 13 0 (0.85869565 0.14130435)
#-#       10) TotalDiscount< 42.08757 81  6 0 (0.92592593 0.07407407) *
#-#       11) TotalDiscount>=42.08757 11  4 1 (0.36363636 0.63636364) *
#-#    3) TotalDiscount>=109.9136 30 14 0 (0.53333333 0.46666667)
#-#      6) TotalDiscount>=146.6379 12  1 0 (0.91666667 0.08333333) *
#-#      7) TotalDiscount< 146.6379 18  5 1 (0.27777778 0.72222222) *

# Here is some of the output explained:
# ?rpart.object
# split= means the predictor split
# n= number of observations reaching the node
# loss= how many observations are for the wrong class
#       (loss=yprob(other class than yval)*n
# yval= if you run a new example through the tree it will have this value
# yprob= the proportion of examples having respectively 0 and 1

#We can also visualize the tree:
par(xpd = TRUE)
plot(tree, compress = TRUE)
text(tree, use.n = TRUE)
```




```
# How to interpret the tree?
# The first split is whether TotalDiscount < 109.9
# If yes go to the left, otherwise go to the right.
# If we go to the right we encounter the same
# variable but now another split is used:
# TotalDiscount >= 146.6. If yes, go to the left.
# If no, go to the right.
# The numbers can be interpreted as follows:
#  -the top number is the predicted label (0 or 1).
#  -and in a/b, a is the number of 0s and b is
#   the number of 1s. For example 11/1 means
```

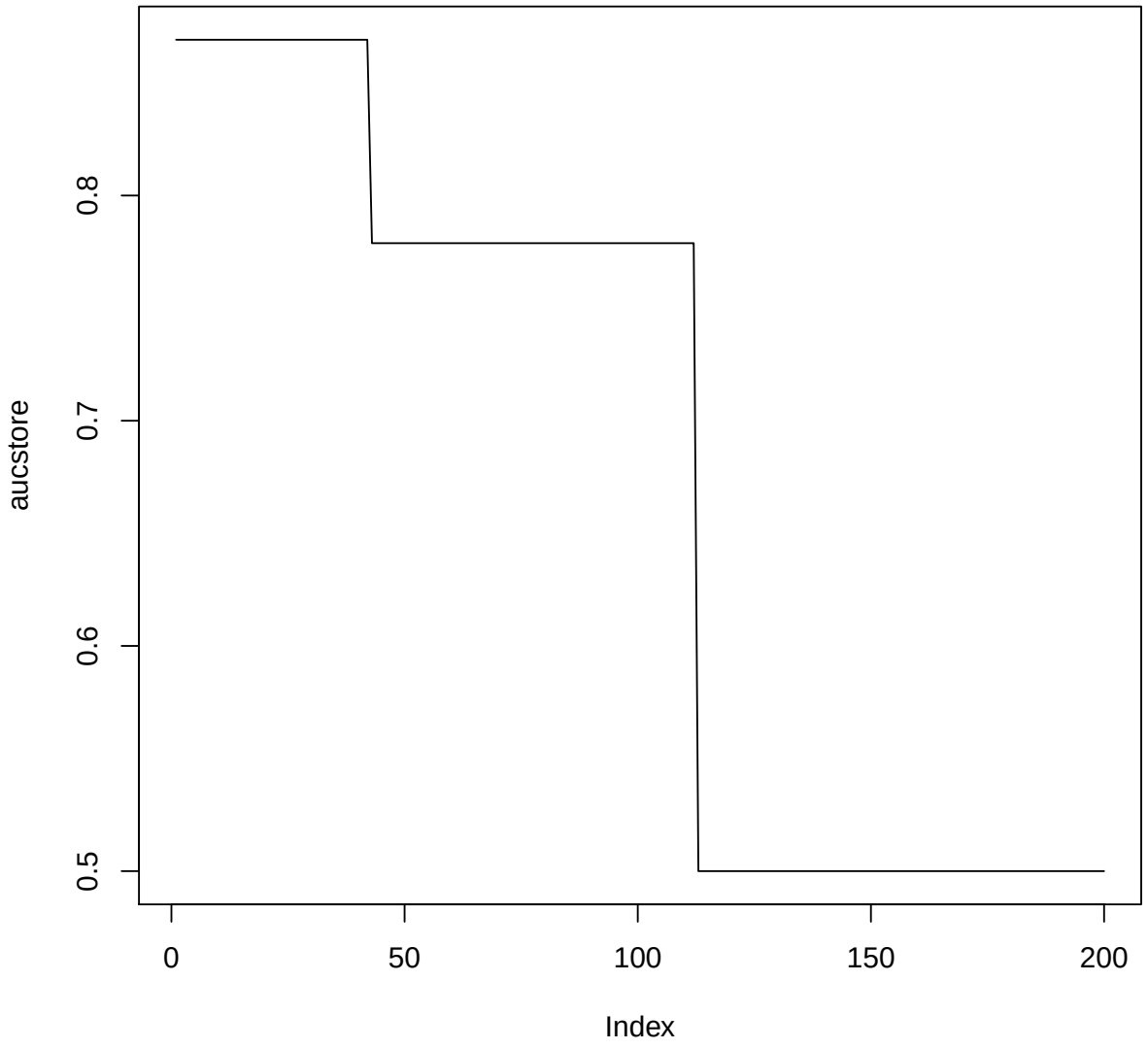
```
#   that there were 11 zeroes and 1 ones.
# Whether the predicted label is 0 or 1 depends on
# whether there are more 0s or 1s.
# The probability of 1 is the proportion of 1s.

#Tuning trees

# It is possible that, given a certain combination of variables,
# there exists no split that reduces the overall lack of fit to a
# sufficient degree. Any split that does not improve the fit by cp
# (complexity parameter) will not be pursued. For example, if the
# fit is not improved by 50%, do not split the tree. This results
# in no splits:
(tree <- rpart(yTRAIN ~ .,
               control=rpart.control(cp = 0.5),
               BasetableTRAIN))

#-# n= 707
#-#
#-# node), split, n, loss, yval, (yprob)
#-#       * denotes terminal node
#-#
#-# 1) root 707 36 0 (0.94908062 0.05091938) *

# This results in one and the same prediction for all observations
# If we make a prediction using that model we will get for all
# observations the proportion of the majority class:

table(yTRAIN)[2]/sum(table(yTRAIN))

#-#          1
#-# 0.05091938

table(predTree <- predict(tree,BasetableTEST)[,2])

#-#
#-# 0.0509193776520509
#-#                707

#So we may decrease the required improvement in fit to get splits
#by decreasing the cp parameter
(tree <- rpart(yTRAIN ~ .,
              control=rpart.control(cp = 0.001),
              BasetableTRAIN))

#-# n= 707
#-#
#-# node), split, n, loss, yval, (yprob)
#-#       * denotes terminal node
#-#
#-#  1) root 707 36 0 (0.94908062 0.05091938)
```

```
#-#    2) TotalDiscount< 109.9136 677 22 0 (0.96750369 0.03249631)
#-#      4) Recency>=21.5 585  9 0 (0.98461538 0.01538462) *
#-#      5) Recency< 21.5 92 13 0 (0.85869565 0.14130435)
#-#       10) TotalDiscount< 42.08757 81  6 0 (0.92592593 0.07407407) *
#-#       11) TotalDiscount>=42.08757 11  4 1 (0.36363636 0.63636364) *
#-#    3) TotalDiscount>=109.9136 30 14 0 (0.53333333 0.46666667)
#-#      6) TotalDiscount>=146.6379 12  1 0 (0.91666667 0.08333333) *
#-#      7) TotalDiscount< 146.6379 18  5 1 (0.27777778 0.72222222) *

table(predTree <- predict(tree,BasetableTEST)[,2])

#-#
#-# 0.0153846153846154 0.0740740740740741 0.0833333333333333
#-#                596                 69                  6
#-#  0.636363636363636  0.722222222222222
#-#                 10                 26

# we see more unique values

# The importance of variables is the improvement in node
# impurity thanks to a variable. In binary classification
# the Gini index is used as measure of node
#impurity: p(1-p)=p-p^2, where p is the proportion of ones.
#The Gini function looks like this:
plot(seq(0,1,0.1),seq(0,1,0.1)-seq(0,1,0.1)^2,type="l",
     ylab="Gini index",xlab="Proportion of ones")
```




```
#This means that Gini is at maximum when a node is equally
# divided amongst both classes

# How to get variable importance for all variables?

#look at the names in the object
names(tree)

#-#  [1] "frame"               "where"
#-#  [3] "call"                "terms"
#-#  [5] "cptable"             "method"
```

```
#-#  [7] "parms"               "control"
#-#  [9] "functions"           "numresp"
#-# [11] "splits"              "variable.importance"
#-# [13] "y"                   "ordered"

#there is an element called variable.importance
tree$variable.importance #higher values means higher importance

#-#  TotalDiscount     TotalPrice      Frequency        Recency
#-#     22.8321535      6.4134062      4.7371592      4.4802579
#-# PaymentType_DD
#-#      0.2466194

#normalize the values:
tree$variable.importance/sum(tree$variable.importance)

#-#  TotalDiscount     TotalPrice      Frequency        Recency
#-#    0.589831869    0.165680008    0.122376870    0.115740237
#-# PaymentType_DD
#-#    0.006371015

#more information can be obtained through:
#summary(tree)

#Let's cross validate the cp parameter

candidates <- seq(0.00001,0.2,by=0.0010)
aucstore <- numeric(length(candidates))
j <- 0
for (i in candidates) {
  j <- j + 1
  tree <- rpart(yTRAIN ~ .,
                control=rpart.control(cp = i),
                BasetableTRAIN)
  predTree <- predict(tree,BasetableVAL)[,2]
  aucstore[j] <- AUC::auc(roc(predTree,yVAL))
  if (j %% 20==0) cat(j/length(candidates)*100,"% finished\n")
}

#-# 10 % finished
#-# 20 % finished
#-# 30 % finished
#-# 40 % finished
#-# 50 % finished
#-# 60 % finished
#-# 70 % finished
#-# 80 % finished
#-# 90 % finished
#-# 100 % finished

plot(aucstore,type="l")
```

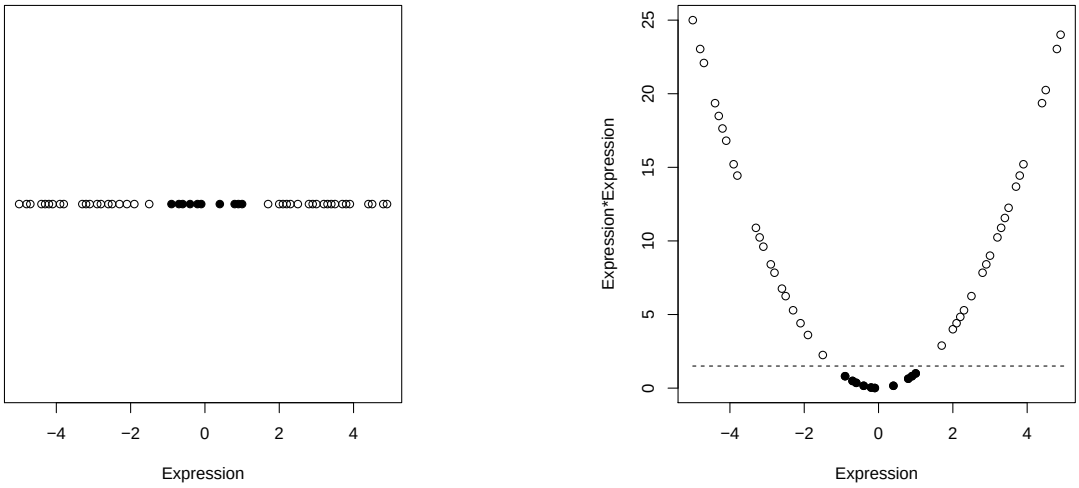



```
#what is the position of the best auc?
which.max(aucstore)

#-# [1] 1

#next we train the model on TRAINbig with the optimal cp
#and confirm final performance on the test set
tree <- rpart(yTRAINbig ~ .,
              control=rpart.control(cp = candidates[which.max(aucstore)]),
              BasetableTRAINbig)
predTREE <- predict(tree,BasetableTEST)[,2]
```

```
#Final model performance:
AUC::auc(roc(predTREE,yTEST))

#-# [1] 0.793545
```

2.5.8 Support vector machines

A major ingredient in Support Vector Machines (SVMs) is the kernel, short for kernel function. Kernels have been introduced to mainstream machine learning literature in 1992 with the paper of Boser et al. (1992) on support vector machines. Before we explain what kernels are, first let's consider the main motivation for using kernels. The left panel in Figure 2.13 (adapted from Noble, 2006) illustrates a one-dimensional data set. The problem is that the dots and circles are inseparable by a single point. By adding an additional dimension to the data (in this case squaring the original expression) the circles and dots become separable by a straight line in the two-dimensional space. The plot on the left is called input space $\mathcal{X}$ and the plot on the right is called feature space or linearization space $\mathcal{F}$.



Inseparable one-dimensional data set Separable data set

Figure 2.13: Motivation for using kernels

Let's drive this point home by showing you that this indeed results in a perfect model.

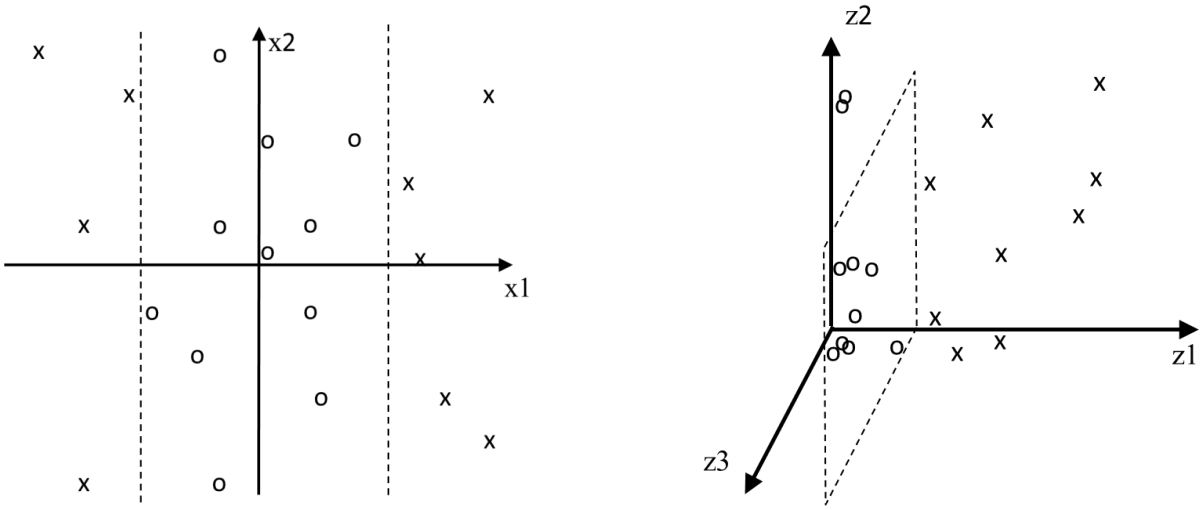
```
# Create data similar to above plots
x <- runif(100, min=-5, max=5)
y <- factor(ifelse(x < 2 & x > -2, 1, 0))

# Check:
# plot(x,rep(1,length(x)), col=y)
# plot(x,x^2, col=y)
```

```
#load the package AUC to evaluate model performance
if(require('AUC')==FALSE)  {
  install.packages('AUC',
                   repos="https://cran.rstudio.com/",
                   quiet=TRUE)
  require('AUC')
}

#-# Loading required package: AUC
#-# AUC 0.3.0
#-# Type AUCNews() to see the change log and ?AUC to get an overview.

# Now, we are going to fit two models.
# The first model corresponds to the plot on the left in the above figure
# The second model corresponds to the plot on the right.

#We use an algorihtm that only fits linear relationships (GLM)

#GLM with x as predictor
LR1 <- glm(y ~ .,
           data=data.frame(x=x),
           family=binomial("logit"))
predLR1 <- predict(LR1,
                   newdata=data.frame(x=x),
                   type="response")
AUC::auc(roc(predLR1,y))

#-# [1] 0.546875

#GLM with x and x^2 as predictors
LR2 <- glm(y ~ .,
           data=data.frame(x=x,xx=x^2),
           family=binomial("logit"))

#-# Warning: glm.fit: algorithm did not converge
#-# Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred

predLR2 <- predict(LR2,
                   newdata=data.frame(x=x,xx=x^2),
                   type="response")
AUC::auc(roc(predLR2,y))

#-# [1] 1

# The second model results in a perfect fit. The only thing we had to do
# is add x^2 to the predictor set.
# Note that glm produces error messages whenever a perfect fit happens.
# This is nothing to worry about when we are doing predictive analytics.
```

Now consider a second example; the twodimensional binary classification problem in the left

pane of Figure 2.14 (adapted from Schölkopf and Smola (2002)). By applying feature map ϕ that transforms the input space (x_1, x_2), by taking all second-degree unordered monomials, a dimension can be found that affords us to separate the data linearly (see right pane of Figure 2.14; note that actually $z_3 = \sqrt{2}x_1x_2$ and not x_1x_2 but this will create a similar plot with the same decision boundary):

$$\begin{aligned} &\phi : \mathbb{R}^2 \to \mathbb{R}^3 \\ &\phi(x_1, x_2) = (z_1, z_2, z_3) = (x_1^2, x_2^2, x_1x_2) \end{aligned} \tag{2.33}$$



Input space $\mathcal{X}$ Feature space $\mathcal{F}$

Figure 2.14: Input space $\mathcal{X}$ and Feature space $\mathcal{F}$

The feature map ϕ transforms the input space (x_1, x_2), by taking all second-degree unordered monomials, to the feature space (z_1, z_2, z_3) where we can classify the data linearly.

Despite the clear advantages of this approach, analysis may soon become intractable. If there are 100 features in the data set instead of one, the resulting data set will contain 5050 features. The computational effort for a subsequent learner scales with the dimensionality (i.e., number of features n) of the new higher dimensional space, also called linearization (Jäkel et al., 2007) or feature space $\mathcal{F}$. We want to work in $\mathcal{F}$ to obtain better models, but it may be too expensive for us to do so.

Consider Equation 2.34 (Schölkopf and Smola, 2002) that shows that the dimensionality n in feature space can easily explode given monomials of degree d, making analysis prohibitive.

$$n_{\mathcal{F}} = \binom{d + n_{\mathcal{X}} - 1}{d} = \frac{(d + n_{\mathcal{X}} - 1)!}{d!(n_{\mathcal{X}} - 1)!} \tag{2.34}$$

where $\mathcal{X}$ denotes input space and $\mathcal{F}$ feature space

In our example $d = 2$ and $n_{\mathcal{X}} = 2$ which results in $n_{\mathcal{F}} = 3$. For $d = 2$ and $n_{\mathcal{X}} = 150$ dimensionality $n_{\mathcal{F}}$ amounts to 11,325. If in the latter case the degree increases by one ($d = 3$) then $n_{\mathcal{F}} = 573,800$.

Kernels offer a solution to this feature explosion so that the computational effort scales linearly with the size (i.e., number of observations N) of the original space, called input space $\mathcal{X}$ (Schölkopf and Smola, 2002). Calculating the inner product, and thus comparing instances as a whole, as opposed to attributes of instances, limits the feature explosion so that the complexity of a classifier increases only linearly with the size of the input data. For two instances x_i and x_j (i,j=1,2,3,...,N) defined by two variables (x_1, x_2):

$$\begin{aligned}\langle\phi(x_{i1}, x_{i2}), \phi(x_{j1}, x_{j2})\rangle &= \langle(x_{i1}^2, x_{i2}^2, \sqrt{2}x_{i1}x_{i2})(x_{j1}^2, x_{j2}^2, \sqrt{2}x_{j1}x_{j2})\rangle \\ &= x_{i1}^2x_{j1}^2 + x_{i2}^2x_{j2}^2 + 2x_{i1}x_{j1}x_{i2}x_{j2} \\ &= (x_{i1}x_{j1} + x_{i2}x_{j2})^2 \\ &= \langle(x_{i1}, x_{i2}), (x_{j1}, x_{j2})\rangle^2 \end{aligned} \tag{2.35}$$

The result in Equation 2.35 shows that the inner product in $\mathcal{F}$ equals the inner product to the power of d in $\mathcal{X}$. This is attractive because whereas the computational effort in $\mathcal{F}$ scales with the number of dimensions (see Equation 2.34), in $\mathcal{X}$ it scales with the number of instances. In the case of a monomial feature map ϕ, it is not required to map the observations x_i and x_j to feature space to compute the inner product: it is sufficient to calculate the inner product in the input space and take it to the power of d. The combination of an inner product and ϕ defines a kernel (Equation 2.36), short for kernel function, k(x_i, x_j). The fact that kernels enable us to obtain the same superior classification performance as in feature space, for a much lower computational cost in input space, is called the kernel trick. We do not need to map to $\mathcal{F}$ first to compute the inner product. We can calculate the inner product in $\mathcal{X}$ and take it to the power of d.

$$k(x_i, x_j) = \langle\phi(x_i), \phi(x_j)\rangle \tag{2.36}$$

Let's summarize what we know about kernels in Figure 2.15. We have four cases. The first column of the matrix denotes all cases in input space, and the second column denotes the cases in feature space. In the top row of the matrix we have cases in which we compare attributes of instances, like we have done in all other algorithms. In the bottom row of the matrix, we use inner products, and therefore we compare instances as a whole. In case 1 we have low performance and low cost. When we move to case 2, we get high performance, but also high cost. In case 3 we use the inner product, but still need to apply the mapping, meaning we still get high performance and high cost. Finally, we move to case 4 in which we not only have low cost, but also high performance as we demonstrated in Equation 2.35 that we do not need to apply the mapping before computing the inner product.

In Figure 2.15 and Equation 2.36 we have been talking about two instances. Let's now have a look at when we have more than two instances.

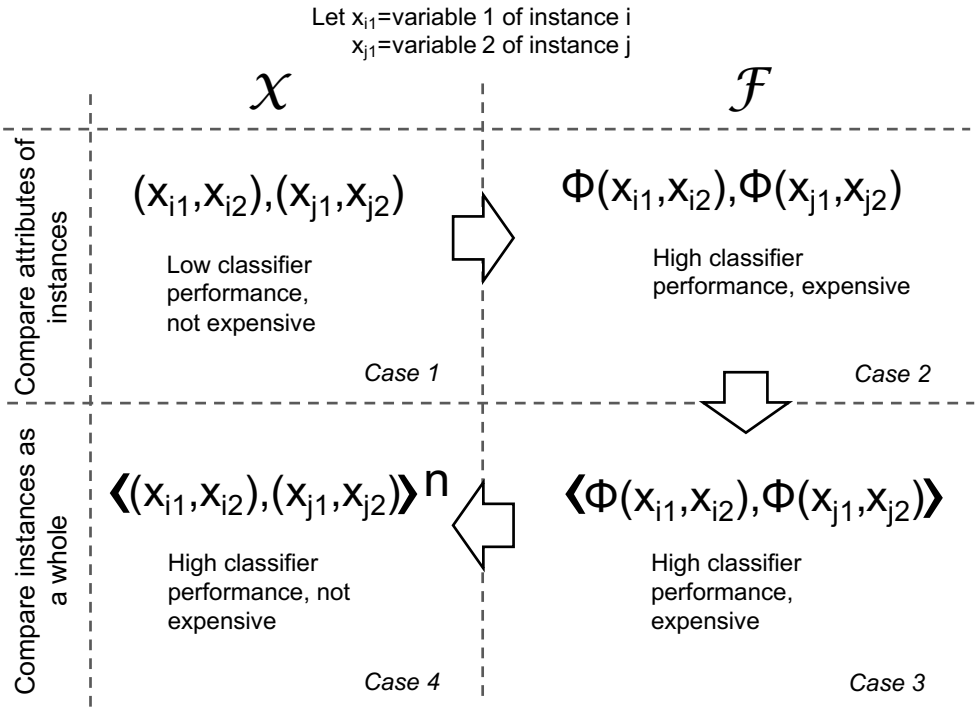


Figure 2.15: Summary of benefits of kernels

```
#case 1
# Suppose we have 3 variables, and 4 instances
case1 <- data.frame(c1=1:4,
                    c2=c(1,2,1,2),
                    c3=c(1,2,1,1))
rownames(case1) <- c('r1','r2','r3','r4')
case1

#-#    c1 c2 c3
#-# r1  1  1  1
#-# r2  2  2  2
#-# r3  3  1  1
#-# r4  4  2  1

#case 2
# (c_1 + c_2+ c_3)^2
# =(c_1 + c_2+ c_3)*(c_1 + c_2+ c_3)
# =c_1*c_1 + c_1*c_2 + c_1 * c_3 +
# c_2*c_1 + c_2*c_2 + c_2 * c_3 +
# c_3*c_1 + c_3*c_2 + c_3*c_3
# =c_1^2 + 2*c1*c2 + c_2^2 + 2*c_1*c_3 + 2*c_2*c_3 + c_3^2
# Suppose again that 2=sqrt(2): because it is a constant,
# it makes no difference

case2 <- data.frame(c_1_square=case1$c1^2,
```

```
                    c_2_square=case1$c2^2,
                    c_3_square=case1$c3^2,
                    c1_c2=sqrt(2)*case1$c1*case1$c2,
                    c1_c3=sqrt(2)*case1$c1*case1$c3,
                    c2_c3=sqrt(2)*case1$c2*case1$c3)
rownames(case2) <- c('r1','r2','r3','r4')
case2

#-#    c_1_square c_2_square c_3_square     c1_c2    c1_c3
#-# r1          1          1          1  1.414214 1.414214
#-# r2          4          4          4  5.656854 5.656854
#-# r3          9          1          1  4.242641 4.242641
#-# r4         16          4          1 11.313708 5.656854
#-#       c2_c3
#-# r1 1.414214
#-# r2 5.656854
#-# r3 1.414214
#-# r4 2.828427

#This results in 4 rows and 6 columns
#If we would have 150 variables, we would get 11,325 columns

#case 3
(as.matrix(case2) %*% as.matrix(t(case2)))

#-#    r1  r2  r3  r4
#-# r1  9  36  25  49
#-# r2 36 144 100 196
#-# r3 25 100 121 225
#-# r4 49 196 225 441

#case 4
(as.matrix(case1) %*% as.matrix(t(case1)))^2

#-#    r1  r2  r3  r4
#-# r1  9  36  25  49
#-# r2 36 144 100 196
#-# r3 25 100 121 225
#-# r4 49 196 225 441

# Conclusion
# Both case 3 and 4 are equivalent, but case 3 requires a lot more computation
# in that it also requires the computations from case 2, whereas the computations
# in case 4 do not require those.
```

Note that the matrix in case 3 and 4 is called a kernel matrix K, also called Gram matrix, and is a matrix that collects all pairwise applications of a kernel function $k(.,.)$ to a set of instances x_1 to x_N. K is a N x N matrix

$$K = \begin{pmatrix} k(x_1, x_1) & k(x_1, x_2) & \cdots & k(x_1, x_N) \\ k(x_2, x_1) & k(x_2, x_2) & \cdots & k(x_2, x_N) \\ \vdots & \vdots & \ddots & \vdots \\ k(x_N, x_1) & k(x_N, x_2) & \cdots & k(x_N, x_N) \end{pmatrix}$$

with an entry in the ith row and jth column denoted as k_{ij}:

$$k_{ij} = k(x_i, x_j) \tag{2.37}$$

Kernels provide a novel data representation in that data are not represented individually anymore, but through a set of pairwise comparisons or similarities (Vert et al., 2004). A kernel matrix acts as an information bottleneck, since all the information available to a kernel learner (e.g., about distribution, model, noise) must be extracted from this matrix (Shawe-Taylor and Cristianini, 2004, p47). It is perhaps surprising how much information about an input data set can be obtained simply from its kernel matrix (Shawe-Taylor and Cristianini, 2004, p138).

One of the main advantages of using kernel functions is that the resulting kernel matrix is always N x N, regardless of the complexity of the instances. This is very attractive when the number of instances is smaller than the number of features that would have been created if they were computed explicitly. For example, a data set with 5000 instances and 150 dimensions, and a polynomial function of degree 2 would result in 11325 dimensions. Using a kernel the dimensionality would only be 5000. Kernels become even more attractive when a relatively small number of complex objects needs to be processed (Vert et al., 2004).

In our examples we use the polynomial kernel function. The polynomial kernel including tuning parameters and two other widely used kernels are displayed in Table 2.8 (Shawe-Taylor and Cristianini, 2004; Park et al., 2012). The choice of the kernel function is largely dependent upon the data and can generally be determined by cross-validation (Fan, 2009).

Table 2.8: Some examples of kernels

Linear kernel	$k(x_i, x_j) = \langle x_i, x_j \rangle$
Gaussian kernel	$k(x_i, x_j) = exp(-\frac{\|x_i - x_j\|^2}{2\sigma})$
Polynomial kernel	$k(x_i, x_j) = (\gamma \langle x_i, x_j \rangle + r)^d$

d,r $\in \mathbb{N}$, $\gamma \in \mathbb{R}^+$

Any algorithm that can be kernelized (i.e., its dependence on the data is only through dot products) can be used with kernels (Ben-Hur and Weston, 2010, p4). The prime example of such an algorithm is Support Vector Machines, displayed in Figure 2.16. The goal of a SVM is to find the maximum margin between support planes. Support planes are plans with points (of either class) on them. The geometrical interpretation of an inner product is the product of the norms of the two vectors (in this case w and the vector connecting the new point and the intersection of the optimal separating hyperplane) and the cosinus of the angle between the two vectors. Since the norm of a vector is always positive the sign (class) of the new instance depends on the cosinus of the angle.

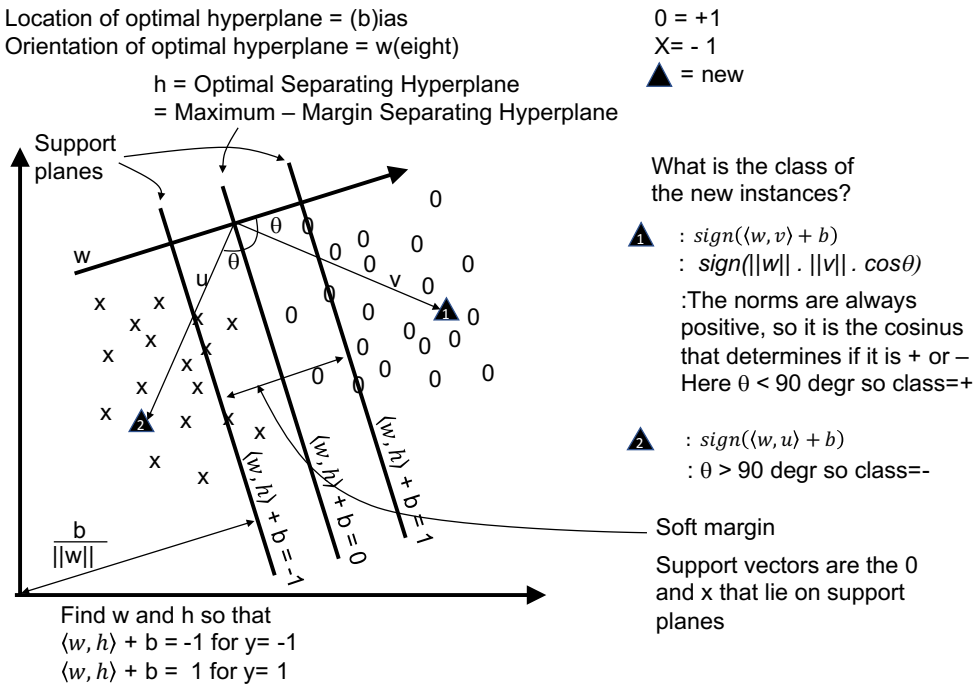


Figure 2.16: Support Vector Machines

An important parameter in SVM is the soft margin constant (Figure 2.17), also called the cost parameter, or C. Smaller values for C allow for errors. It answers the question: 'How much do errors cost?'. The C parameter changes the orientation of the hyperplane and the size of the margin. When points are allowed in the margin, we speak of a soft margin, otherwise we speak of a hard margin. The goal is to find the right value for C. If C is too high, then we are likely to overfit the data. If it is too low, we underfit the data. Cross-validation will tell us the right value for our data.

Now let's have a look at how we can estimate and tune a SVM in R. All code in this section is available at:
`http://ballings.co/hidden/aCRM/code/chapter2/ModelingSVM.R`

```
#Download the data
rm(list=ls())
source("http://ballings.co/hidden/aCRM/code/chapter2/read_data_sets.R")
ls()

#-# [1] "BasetableTEST"     "BasetableTRAIN"
#-# [3] "BasetableTRAINbig" "BasetableVAL"
#-# [5] "classes"           "yTEST"
#-# [7] "yTRAIN"            "yTRAINbig"
#-# [9] "yVAL"

if (require("e1071")==FALSE) {
```

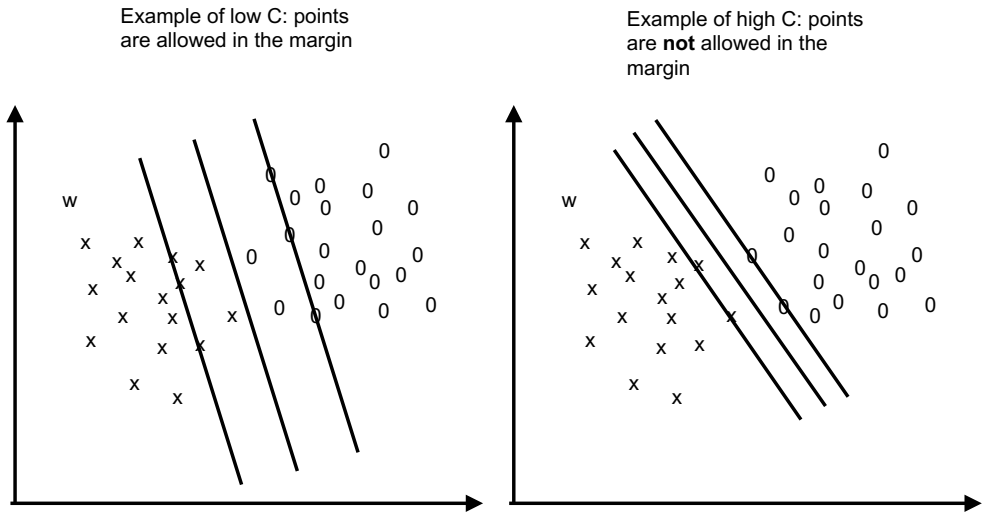


Figure 2.17: The cost parameter in SVM

```
  install.packages("e1071",
                   repos="https://cran.rstudio.com/",
                   quiet=TRUE)
  require(e1071)
}
#load the package AUC to evaluate model performance
if(require('AUC')==FALSE)  {
  install.packages('AUC',
                   repos="https://cran.rstudio.com/",
                   quiet=TRUE)
  require('AUC')
}


#Run an external function to facilitate tuning
source("http://ballings.co/hidden/aCRM/code/chapter2/tuneMember.R")

#tuning grid

SV.cost <- 2^(-5:-4)   #This is good range but takes a long time: 2^(-5:13)
SV.gamma <-  2^(-15:-14)  #This is good range but takes a long time: 2^(-15:3)
# For the polynomial kernel function gamma denotes scale (not very important).
# For the radial basis kernel gamma denotes the inverse kernel width (very important).
SV.degree <-  c(1,2,3)
SV.kernel <- c('radial','polynomial')

result <- list()

for (i in SV.kernel) {
          call <- call("svm",
                       formula = as.factor(yTRAIN) ~ .,
```

```
                        data=BasetableTRAIN,
                        type = "C-classification",
                        probability=TRUE)
          #the probability model for classification fits a logistic distribution using
          #maximum likelihood to the decision values of the binary classifier

          if (i=='radial'  ) tuning <- list(gamma = SV.gamma,
                                            cost = SV.cost,
                                            kernel='radial')
          if (i=='polynomial') tuning <- list(gamma = SV.gamma,
                                              cost = SV.cost,
                                              degree=SV.degree,
                                              kernel='polynomial')

          #tune svm
          result[[i]] <- tuneMember(call=call,
                                    tuning=tuning,
                                    xtest=BasetableVAL,
                                    ytest=yVAL,
                                    probability=TRUE)

}
result

#-# $radial
#-#          gamma    cost kernel      auc
#-# 2 6.103516e-05 0.03125 radial 0.794235
#-#
#-# $polynomial
#-#          gamma    cost degree     kernel       auc
#-# 1 3.051758e-05 0.03125      1 polynomial 0.7852522

# Common error message:
# If you get the The warning "reaching max number of iterations" means that the iterative
# routine used by LIBSVM (which is used by svm) to
# solve quadratic optimization problem in order to find the maximum margin hyperplane
# (i.e., parameters w and b) separating your data reached the maximum number of iterations
# and will have to stop, while the current approximation for w can
# be further enhanced (i.e., w can be changed to make the value of the objective function
# more extreme).
# In short, that means the LIBSVM thinks it failed to find the maximum margin hyperplane,
# which may or may not be true.

# There are many reasons why this may happen, Things you could try:
#
# -Normalize your data.
# -Make sure your classes are more or less balanced (have similar size).
#  If they don't, use parameter -w to assign them different weights.
# -Try different C and 'gamma'. Polynomial kernel in LIBSVM also has parameter 'coef0',
#  as the kernel is (gamma*u'*v + coeff0)^degree.
```

```
#select the best set of parameters
auc <- numeric()
for (i in 1:length(result)) auc[i] <- result[[i]]$auc

result <- result[[which.max(auc)]]


#now predict using these optimal parameters
SV <- svm(as.factor(yTRAIN) ~ .,
          data = BasetableTRAIN,
          type = "C-classification",
          kernel = as.character(result$kernel),
          degree= if (is.null(result$degree)) 3 else result$degree,
          cost = result$cost,
          gamma = if (is.null(result$gamma)) 1 / ncol(BasetableTRAINbig) else result$gamma,
          probability=TRUE)

#Compute the predictions for the test set
predSV <- as.numeric(attr(predict(SV,BasetableTEST, probability=TRUE),"probabilities")[,"1"])

head(predSV)

#-# [1] 0.04809388 0.04652274 0.04723145 0.04872022 0.04539469
#-# [6] 0.04390422

AUC::auc(roc(predSV,yTEST))

#-# [1] 0.7241366
```

2.5.9 Ensemble Methods

Ensembling refers to combining multiple diverse models into one aggregate model. Many diverse classifiers, can be learned from making specific combinations of input data, representation languages, evaluation functions and optimization methods. The resulting set of all possible classifiers that might be learned is called the hypothesis space $\mathcal{H}$. Consider the hypothesis space denoted by the outer curve in the left panel of Figure 2.18 (adapted from Dietterich, 2000). A learner searches the space $\mathcal{H}$ to identify the best hypothesis, $h \in \mathcal{H}$. The inner curve depicts a set of accurate hypotheses. Since one only disposes of a limited amount of data, it is possible that the learner finds different hypotheses (h_1, h_2, h_3) of equal *overall* accuracy. If in some region of the input domain, some hypotheses perform better than others, averaging reduces the risk of selecting the wrong hypothesis in a specific region. The point labeled g is the average of those three hypotheses and f is the true hypothesis. Point g is closer to f than any of the constituent hypotheses meaning that the combination of those points reduces the risk of selecting the wrong classification model. In this case combining or ensembling solves an evaluation problem (called a statistical problem by Dietterich, 2000). It has to be noted that there is no guarantee that the ensemble improves upon the best constituent classifier (Fumera and Roli, 2005). However,

combining models does reduce the probability of selecting a poor model.

The middle panel in Figure 2.18 refers to how ensembling may solve optimization problems (called computational problems by Dietterich, 2000). Learners employ optimization methods for local searches (e.g., trees use greedy search) that may get stuck in local optima. When there are sufficient data and the evaluation problem does not exist, an algorithm may still struggle to find the best hypothesis. By averaging different hypotheses, obtained by starting local searches from different points, the true hypothesis f may be better approximated than any of the individual hypotheses.

The third problem that ensemble methods can alleviate is representational in nature (Dietterich, 2000). When $f \notin \mathcal{H}$, averaging can expand the space of representable functions as depicted in the right panel of Figure 2.18. For example, a linear learner cannot learn non-linear boundaries, while a combination of linear learners can learn any boundary. In essence, an ensemble effectively follows a divide and conquer approach in that each individual only learns a smaller or simpler partition of the problem. Note that very flexible learners such as neural networks and trees can represent all possible classifiers (i.e., f always falls in $\mathcal{H}$). However, this capability only holds asymptotically. In reality only a finite set of hypotheses can be generated given a finite input data set (Dietterich, 2000).

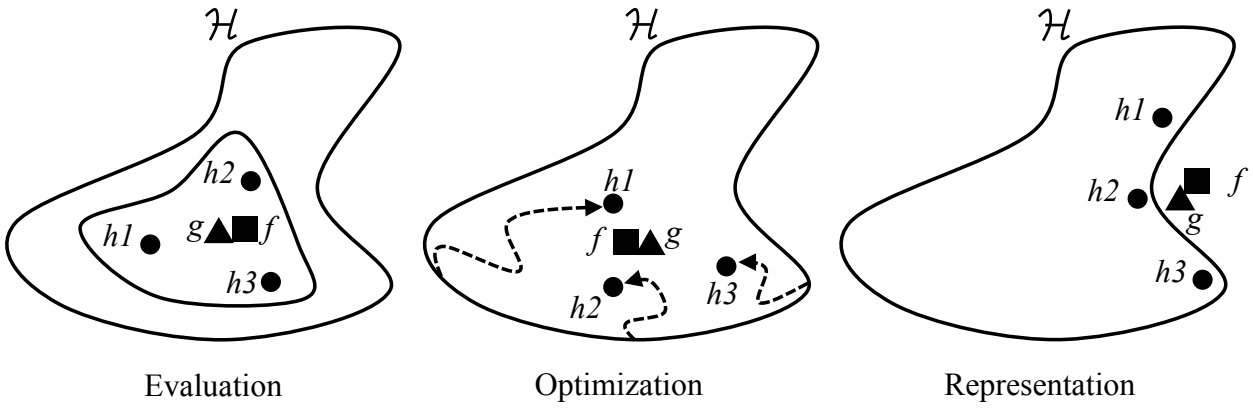


Figure 2.18: Reasons why an ensemble may outperform a single model

Computational problems such as data size limitations and execution time constraints, not depicted in Figure 2.18, can also be alleviated by ensemble methods. Instead of learning one model on the whole data set, multiple smaller models can be learned on different smaller subsets, as such reducing computer hardware requirements. Processing speed can also be increased since most ensembles are inherently parallel. An algorithm that suffers from the evaluation issue or the optimization issue is said to have high variance. An algorithm suffering from the representational issue is said to have high bias. Hence, ensembling may reduce both variance and bias (Zhou, 2012, p67).

The three issues that are exploited by ensembles (evaluation, optimization and representation issues) to deliver superior performance are part of what is called the algorithm diversity strategy, the first strategy of two. For example, one could create a tree k times, each time using a different complexity parameter, and aggregate the trees. Another example is to create multiple

neural networks, each time starting with a different set of random weights. The second strategy is the data diversity strategy. For example, one could create a tree k times, each time using a slightly modified dataset, and aggregate the trees. In the following five sections we present several ensemble methods.

2.5.10 Bagged trees

Bagging stands for bootstrap aggregating and is a type of ensemble model. Ensembles are combinations of multiple diverse models and almost always outperform the best model in the ensemble. A bootstrap sample is created with sampling with replacement, and is of equal size as the original sample. It is a method of data perturbation and diversity generation. Because trees are very sensitive to small changes in the data, building multiple trees and averaging the predictions can yield drastic improvements in predictive performance. Figures 2.19 and 2.20 show the estimation and deployment phase of a bagged tree with ensemble size 3. In the following paragraphs we will provide an intuitive explanation as to how such an approach can lead to improved performance.

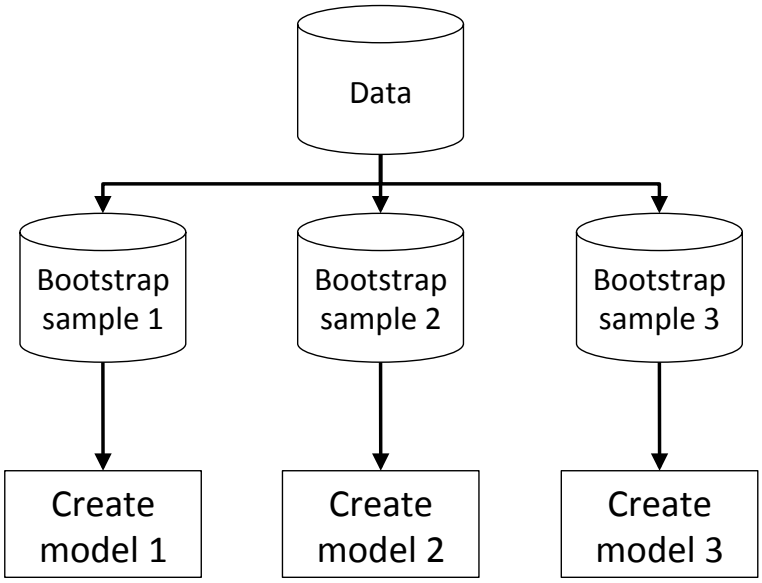


Figure 2.19: Model estimation phase of a bagged tree

This code chunk is available from:
http://ballings.co/hidden/aCRM/code/chapter2/ModelingBT.R

```
#Download the data
rm(list=ls())
source("http://ballings.co/hidden/aCRM/code/chapter2/read_data_sets.R")

#check what is in our environment
ls()
```

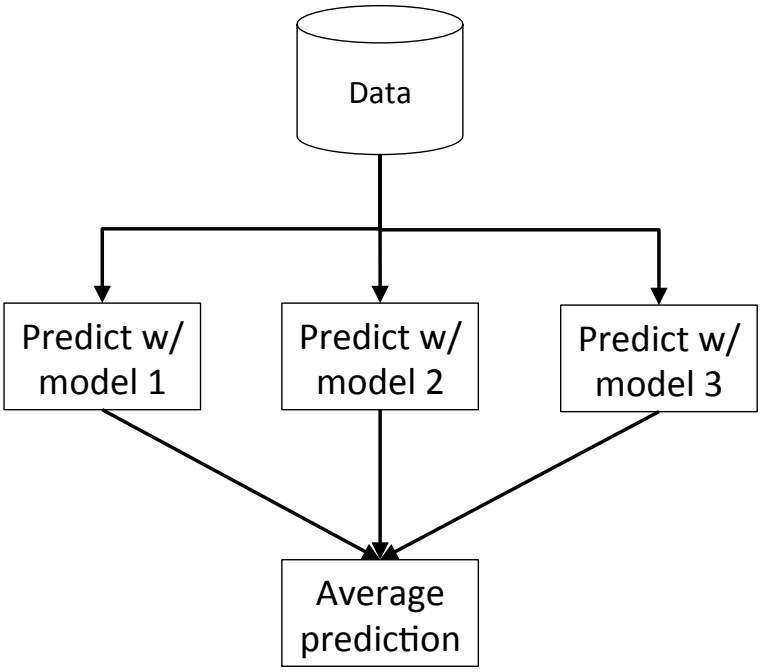


Figure 2.20: Model deployment phase of a bagged tree

```
#-# [1] "BasetableTEST"     "BasetableTRAIN"
#-# [3] "BasetableTRAINbig" "BasetableVAL"
#-# [5] "classes"           "yTEST"
#-# [7] "yTRAIN"            "yTRAINbig"
#-# [9] "yVAL"

#load the package AUC to evaluate model performance
if(require('AUC')==FALSE)  {
  install.packages('AUC',
                   repos="https://cran.rstudio.com/",
                   quiet=TRUE)
  require('AUC')
}

#load the package rpart to create decision  trees
if(require('rpart')==FALSE)  {
  install.packages('rpart',
                   repos="https://cran.rstudio.com/",
                   quiet=TRUE)
  require('rpart')
}

#Create one tree:
tree <- rpart(yTRAIN ~ ., BasetableTRAIN)
predTREE <- predict(tree,BasetableTEST)[,2]

AUC::auc(roc(predTREE,yTEST))
```

```
#-# [1] 0.7818622

#Create 100 trees
ensemblesize <- 100

ensembleoftrees <- vector(mode='list',length=ensemblesize)

for (i in 1:ensemblesize){
    bootstrapsampleindicators <- sample.int(n=nrow(BasetableTRAIN),
                                            size=nrow(BasetableTRAIN),
                                            replace=TRUE)
    ensembleoftrees[[i]] <- rpart(yTRAIN[bootstrapsampleindicators] ~ .,
                                  BasetableTRAIN[bootstrapsampleindicators,])
}


baggedpredictions <- data.frame(matrix(NA,ncol=ensemblesize,
                                       nrow=nrow(BasetableTEST)))

for (i in 1:ensemblesize){

  baggedpredictions[,i] <- as.numeric(predict(ensembleoftrees[[i]],
                                              BasetableTEST)[,2])
}

finalprediction <- rowMeans(baggedpredictions)

AUC::auc(roc(finalprediction,yTEST))

#-# [1] 0.897333

#drastic improvement
```

2.5.11 Random forest

Random forests (Breiman, 2001) cope with the limited robustness and sub-optimal performance (Dudoit et al., 2002) of decision trees by building an ensemble of trees (e.g., a thousand trees) (Breiman, 2001).

Each individual tree is grown on a bootstrap sample using Binary Recursive Partitioning (BRP) (Breiman et al., 1984). In random forests, the BRP algorithm starts by randomly selecting a subset of candidate variables (Breiman, 2001) and evaluating all possible splits of all candidate variables. A binary partitioning of the data is then created using the best split. The algorithm proceeds recursively by, within each parent partition, again randomly selecting a subset of variables, evaluating all possible splits, and creating two child partitions based on the best split. In sum, random forests uses random feature selection at each node of each tree, and each tree is built on a bootstrap sample.

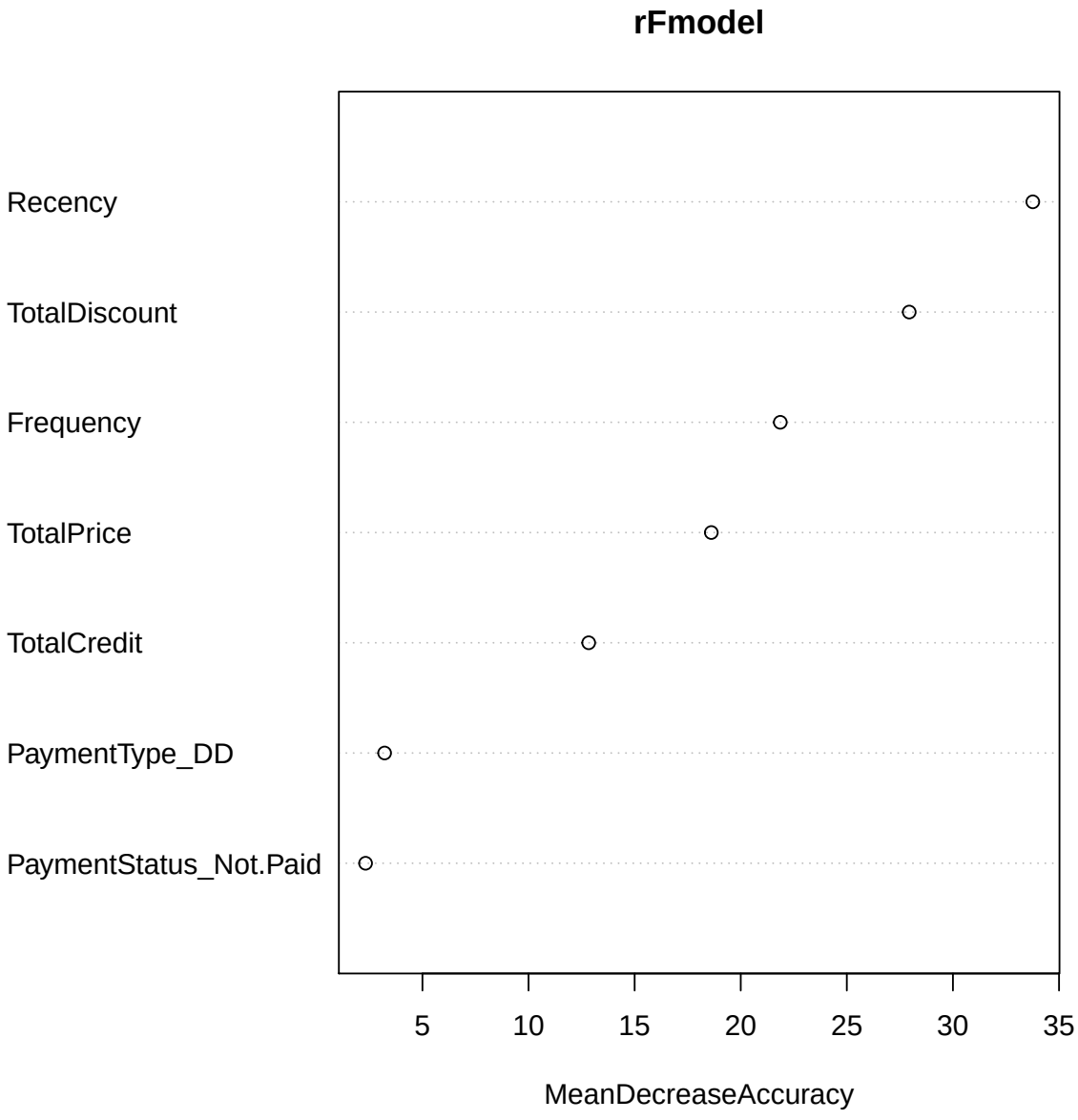
The advantages of Random Forests are manifold. First, literature shows that Random Forests

is one of the best-performing techniques available (Luo et al., 2004; Caruana and Niculescu-Mizil, 2006; Zhou, 2012, p21) and is very robust and consistent (Breiman, 2001). This statement is substantiated in a recent study by Fernandez-Delgado et al. (2014) in which the authors evaluate 179 algorithms arising from 17 families on 121 data sets. They found that Random Forests is most likely to be the best algorithm. Second, the method does not overfit (Breiman, 2001). Third, variable importance measures are available for all variables (Ishwaran et al., 2004). Fourth, the algorithm has reasonable computing times (Breiman, 2001). Finally, the procedure is easy to implement: only two parameters are to be set (number of trees and number of variables) (Larivière and Van den Poel, 2005). We follow the recommendations of Breiman (Breiman, 2001) and use a large number of trees (1000) and the $sqrt(p)$ as the size of the variable subsets where p is the total number of variables. We used the *R* package *randomForest* by Liaw and Wiener (2012) to implement the algorithm.

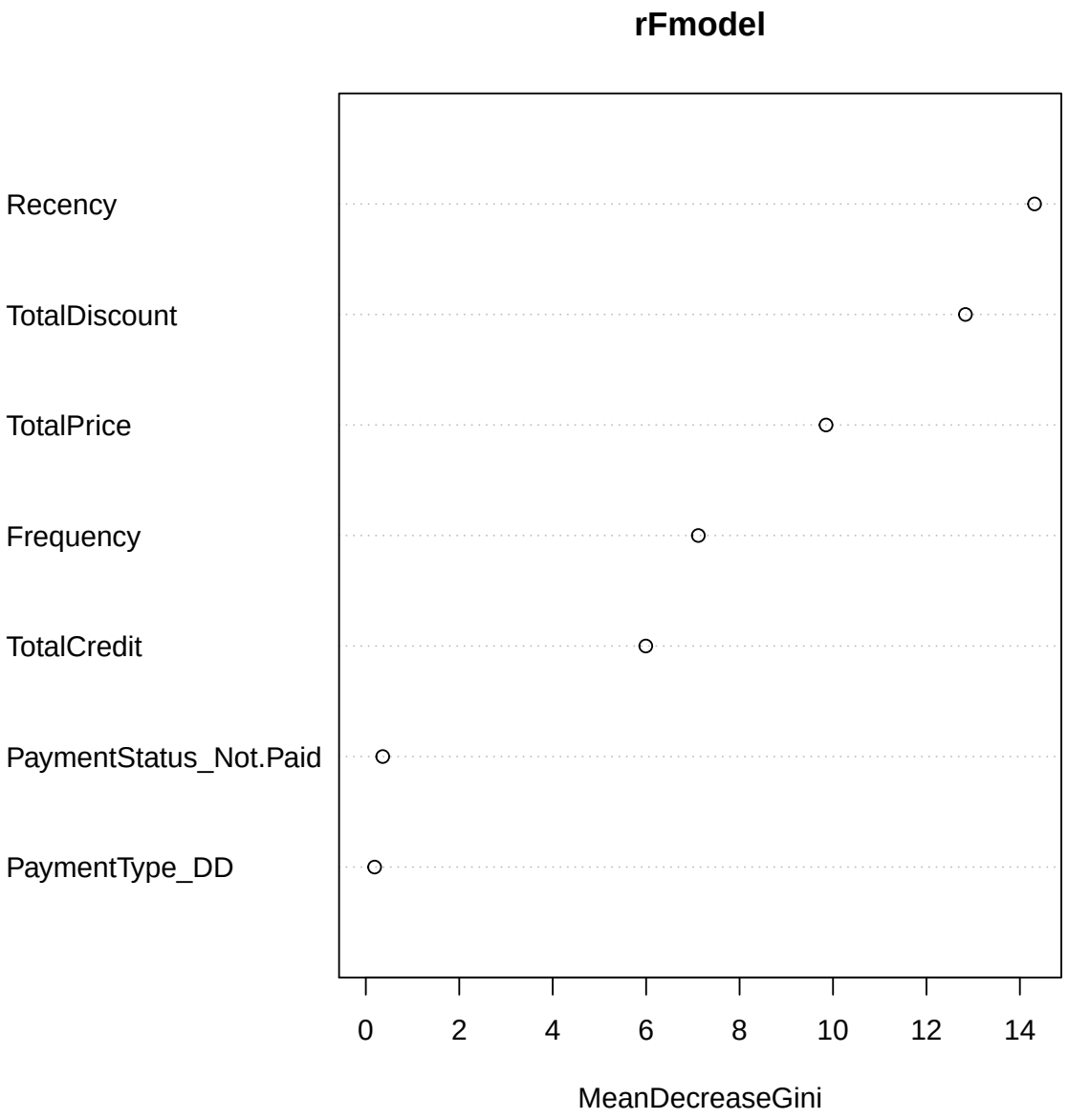
All code in this section is available at:
`http://ballings.co/hidden/aCRM/code/chapter2/ModelingRF.R`

```
source("http://ballings.co/hidden/aCRM/code/chapter2/read_data_sets.R")

#load the package AUC to evaluate model performance
if(require('AUC')==FALSE)  {
  install.packages('AUC',
                   repos="https://cran.rstudio.com/",
                   quiet=TRUE)
  require('AUC')
}

#load the package randomForest
if (!require("randomForest")) {
  install.packages('randomForest',
                   repos="https://cran.rstudio.com/",
                   quiet=TRUE)
  require('randomForest')
}

#always first look at the documentation
?randomForest

#create a first random forest model
rFmodel <- randomForest(x=BasetableTRAIN,
                        y=yTRAIN,
                        ntree=1000,
                        importance=TRUE)

#look at the importance of the variables
importance(rFmodel, type=1)

#-#                       MeanDecreaseAccuracy
#-# TotalDiscount                     27.940401
#-# TotalPrice                        18.613409
#-# TotalCredit                       12.835781
```

```
#-# PaymentType_DD                    3.217591
#-# PaymentStatus_Not.Paid            2.319713
#-# Frequency                        21.864212
#-# Recency                          33.763753

varImpPlot(rFmodel,type=1)
```

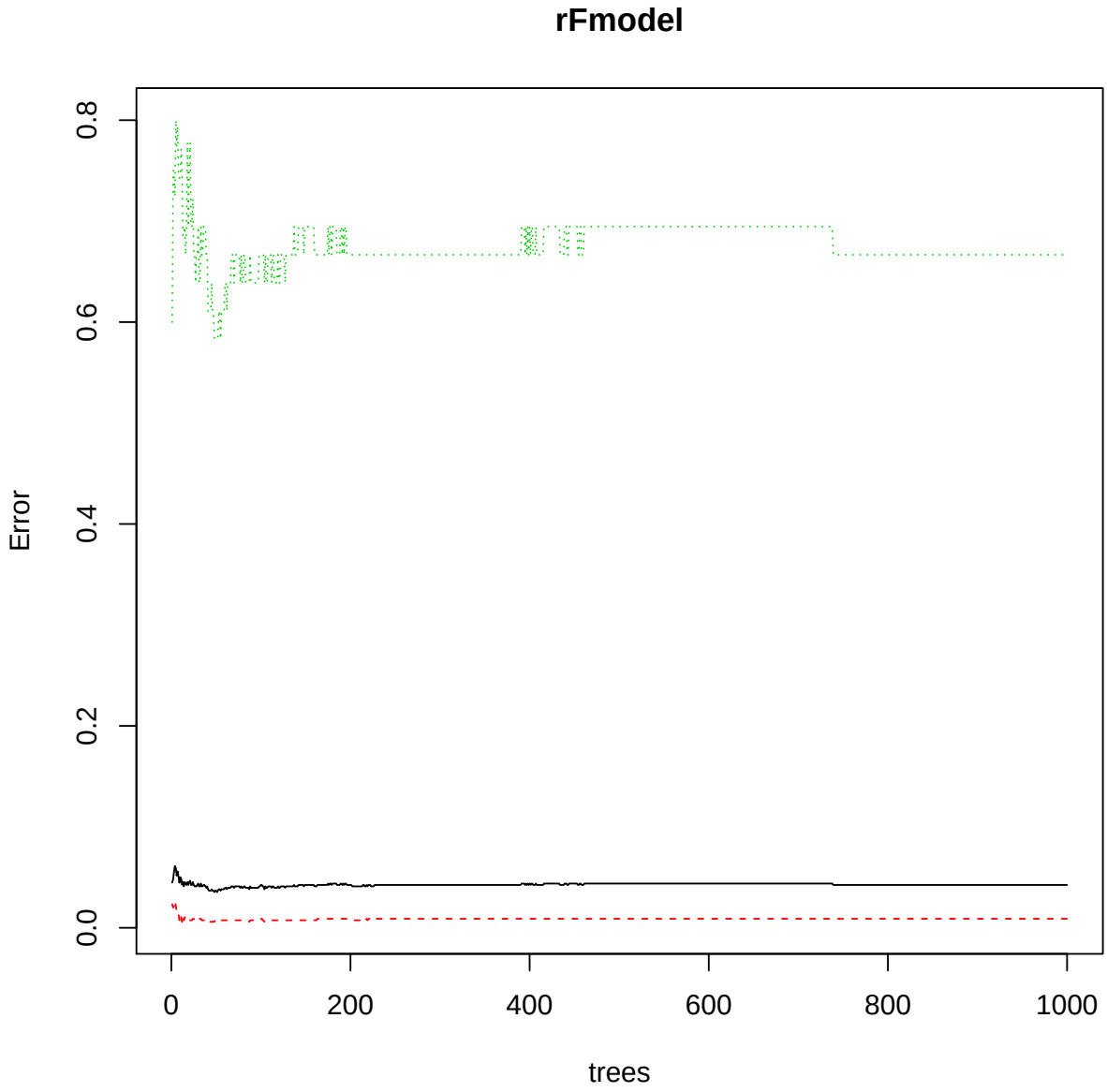



```
varImpPlot(rFmodel,type=2)
```




```
#For more imformation about the measures:
?importance

#prediction
#firt look at the documentation
?predict.randomForest
predrF <- predict(rFmodel,BasetableTEST,type="prob")[,2]
#assess final performance
AUC::auc(roc(predrF,yTEST))

#-# [1] 0.9184171
```

```
#What is the optimal value for ntree

#plotting learing curve

rFmodel <- randomForest(x=BasetableTRAIN,
                        y=yTRAIN,
                        xtest=BasetableVAL,
                        ytest=yVAL,
                        ntree=1000,
                        importance=TRUE)
head(plot(rFmodel))
```



```
#-#             OOB          0         1
#-# [1,] 0.04477612 0.02325581 0.6000000
#-# [2,] 0.04796163 0.01995012 0.7500000
#-# [3,] 0.05566219 0.02213280 0.7500000
#-# [4,] 0.06122449 0.02683363 0.7241379
#-# [5,] 0.05920000 0.02184874 0.8000000
#-# [6,] 0.05167173 0.01594896 0.7741935

#red dashed line: class error 0
#green dotted line: class error 1
#black solid line: OOB error
#we see that class 0 has lower error than class 1.
#This is because there are much more zeroes to learn from.

#create the model with the optimal ensemble size
rFmodel <- randomForest(x=BasetableTRAINbig,
                        y=yTRAINbig,
                        ntree=which.min(rFmodel$test$err.rate[,1]),
                        importance=TRUE)
predrF <- predict(rFmodel,BasetableTEST,type="prob")[,2]
#Final performance
AUC::auc(roc(predrF,yTEST))

#-# [1] 0.9187318

# As we can tell, performance is unexpectely
# a lot lower than the model with 1000 trees.
# This suggests that we cannot accurately tune
# the ntree parameter. The underlying reason
# is that each random forest model is substantially
# different because of the inherent randomization.
# Therefore, never try to tune the number of
# trees. A good approach is to try two models,
# one with 500 trees and one with 1000, and go
# with the 500 trees if they are not substantially
# worse than the 1000 trees if computing speed
# is of the essence.
```

2.5.12 Kernel factory

2.5.13 Adaptive boosting

Boosting is an algorithm that combines many weak classifiers to generate a strong ensemble. It was first introduced by Schapire (1990). The algorithm received important updates by contributions from Freund (1995) and Freund and Schapire (1996). Around the same time Breiman introduced bagging and random forest (Breiman, 1996, 1999, 2001), in which randomness plays a crucial role. Inspired by Breiman (1999), Friedman (2002) made a minor modification to the boosting algorithm to incorporate randomness improving its performance even more. This latter version of

boosting is called stochastic boosting. Stochastic boosting works as follows, based on Friedman et al. (2000); Friedman (2002):

1. Start with giving each instance a weight $w_i = 1/N$, $i = 1, 2, ..., N$, with N the number of instances. Note that $y \in$ {-1,1}.

2. Repeat for $m = 1, 2, ..., M$, with M the number of models (i.e., ensemble size):

   (a) Generate a bootstrap sample (i.e., with replacement) from the full training sample, with the probability of an observation being selected proportional to its most recent weight.

   (b) Fit the classifier to the bootstrap sample and make a prediction $P_m(x) \in$ [0,1], short for $P_m(y = 1 \mid x)$, on that sample

   (c) Compute performance of the model (e.g., AUC), called α_m

   (d) Compute $f_m(x) \leftarrow \frac{1}{2} log(P_m(x)/(1 - P_m(x)))$

   (e) Compute $w_i \leftarrow w_i \; exp(-y \; f_m(x_i))$, and renormalize such that $\sum_{i=1}^{N} w_i = 1$

3. Normalize α_m as follows: $\frac{\alpha_m}{\sum_{i=1}^{N} \alpha_m}$

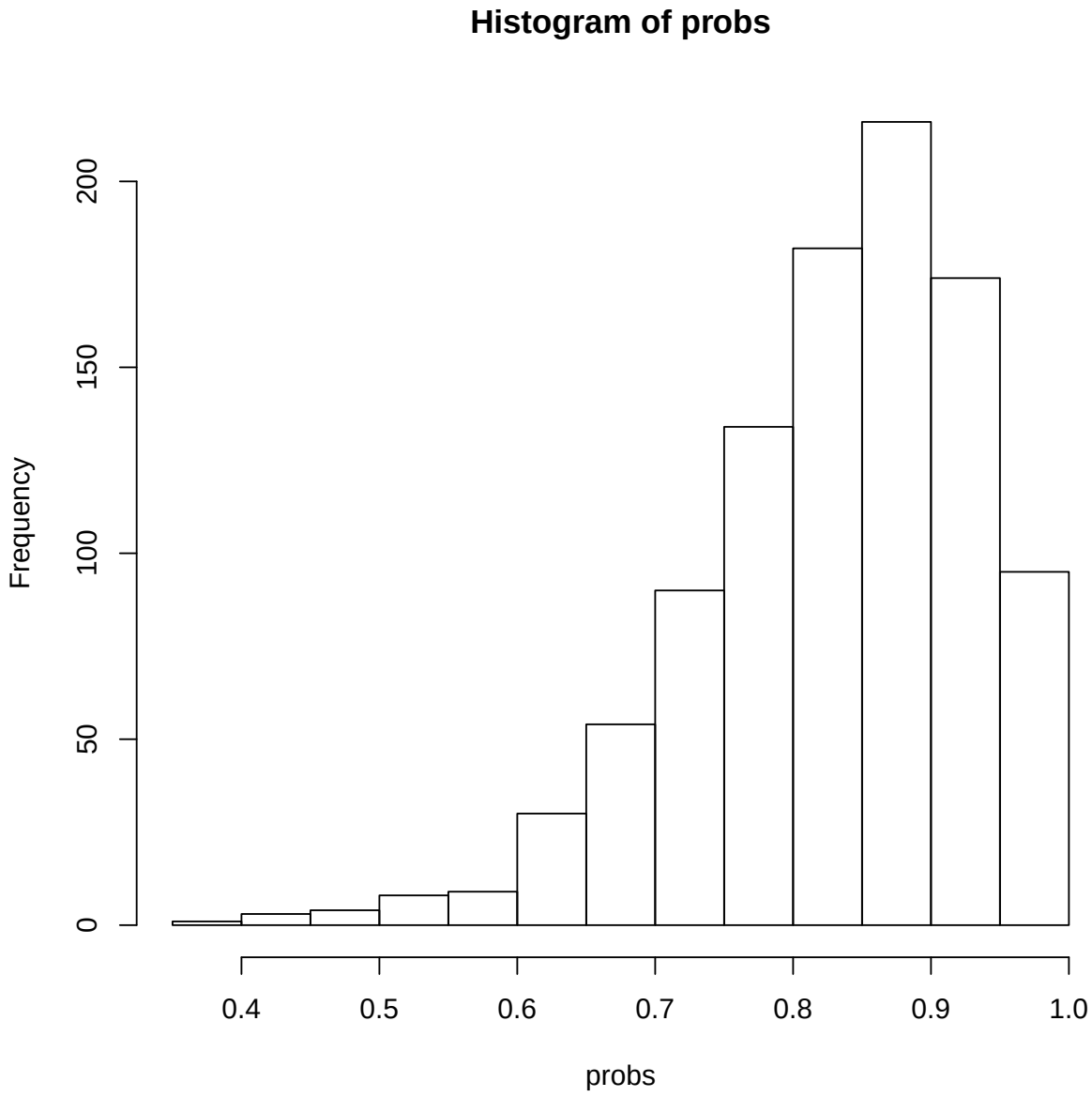
4. Compute the weighted sum of the predictions of the different models: $P(x) = \hat{y} = \sum_{m=1}^{M} \alpha_m P_m(x)$

There are two lines in the algorihtm that require some more explanation: 2(d) and 2(e). In 2(d) we take the logit of the probability. We do this to make the probabilities fit a normal distribution. We want that because we will be adding the predictions of different models in step 4, and different models may have different distributions for the predictions. At this point we are still wondering where the $1/2$ comes from in 2(d). Instead of following the convention to take one class as reference category (i.e., either 0 or 1) the simple mean of the predictions of the categories is used as a reference. The response variable units are then deviations from the mean or the disparity between the logged predictions for category 1 and the mean of the logged predictions for all categories (0 and 1). The units are logits but with the mean over the two categories as reference. In this case using the mean as reference has no advantage, and therefore we accept it as a convention. To see how this plays out, we consider Equation 2.38:

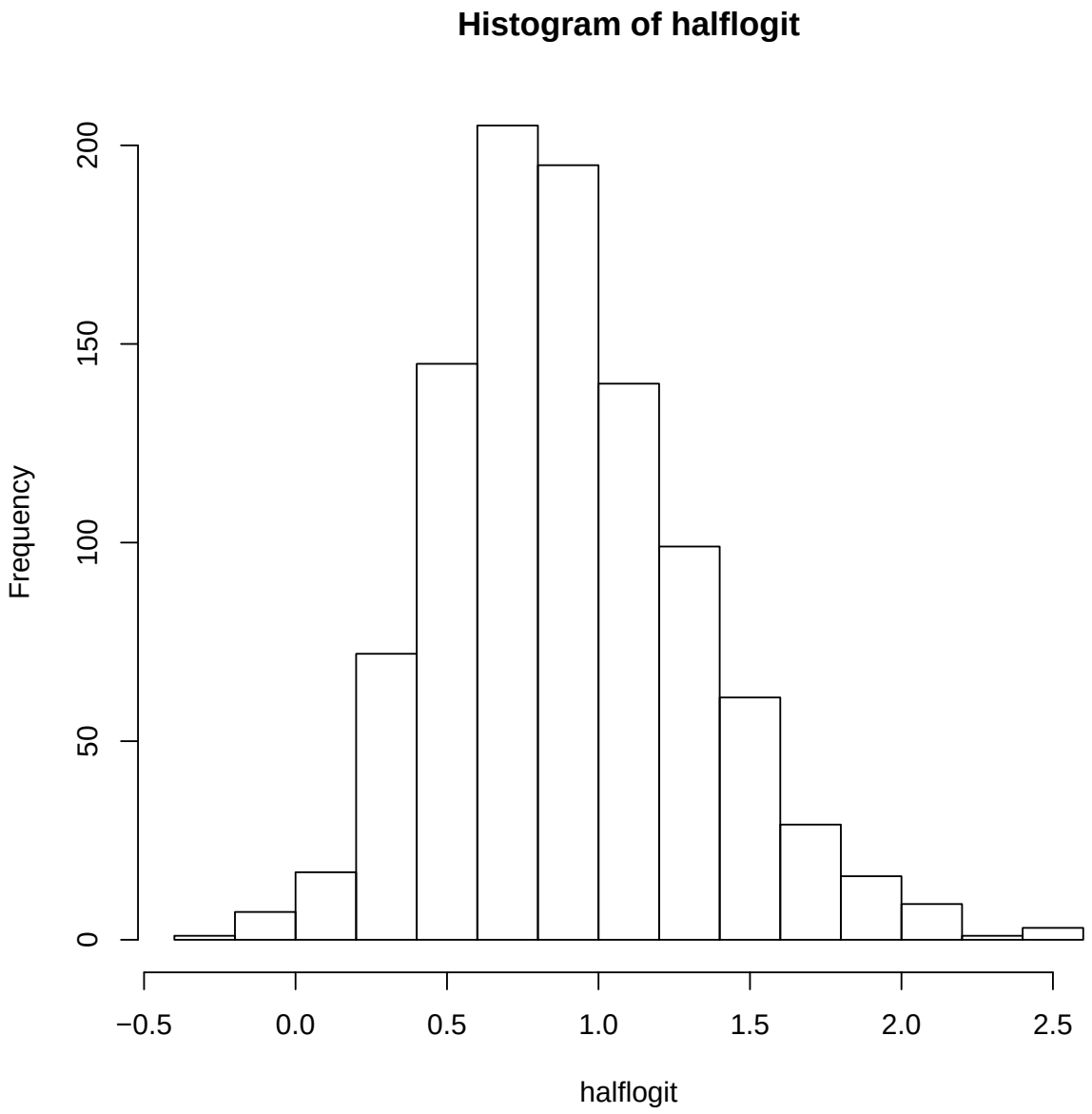
$$
\begin{aligned}
f_m(x) &= \frac{1}{2}\Big(logit\big(P_m(x)\big)\Big) \\
&= \frac{1}{2}\Big(log\big(P_m(x)/(1-P_m(x))\big)\Big) \\
&= \frac{1}{2}\Big(log\big(P_m(x)\big) - log\big(1-P_m(x)\big)\Big) \\
&= \frac{1}{2}log\big(P_m(x)\big) - \frac{1}{2}log\big(1-P_m(x)\big) \\
&= log\big(P_m(x)\big) - \frac{1}{2}log\big(P_m(x)\big) - \frac{1}{2}log\big(1-P_m(x)\big) \\
&= log\big(P_m(x)\big) - \frac{1}{2}\Big(log\big(P_m(x)\big) + log\big(1-P_m(x)\big)\Big)
\end{aligned}
\tag{2.38}
$$

and indeed we find that the last part of the last equation is indeed the mean of the two class predictions. Now, let's prove to ourselves that the logit can indeed make a skewed distribution fit a normal distrubtion.

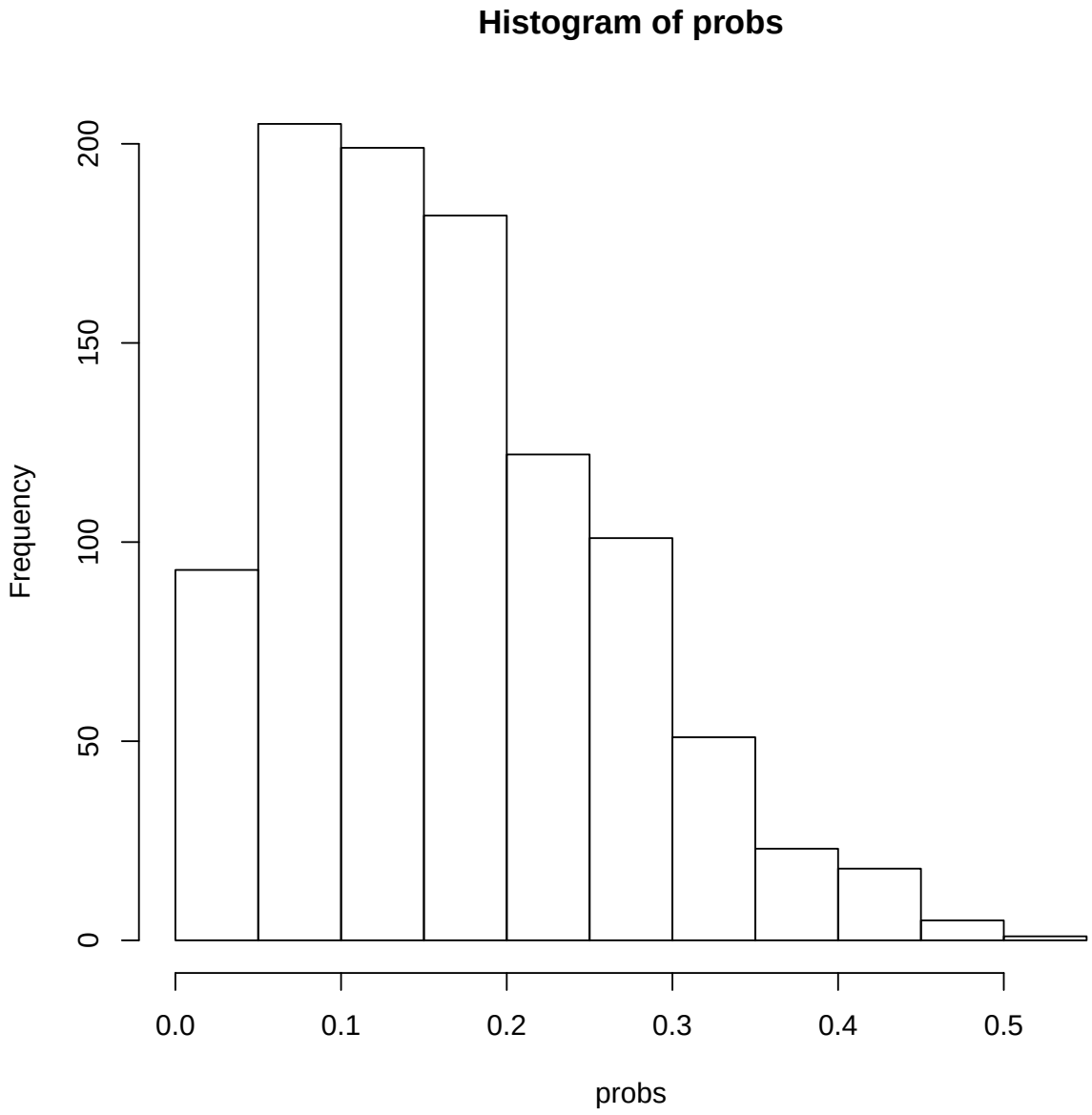
```
#left skewed distribution (large tail to the left)
probs <- rbeta(1000,10,2)
hist(probs)
```



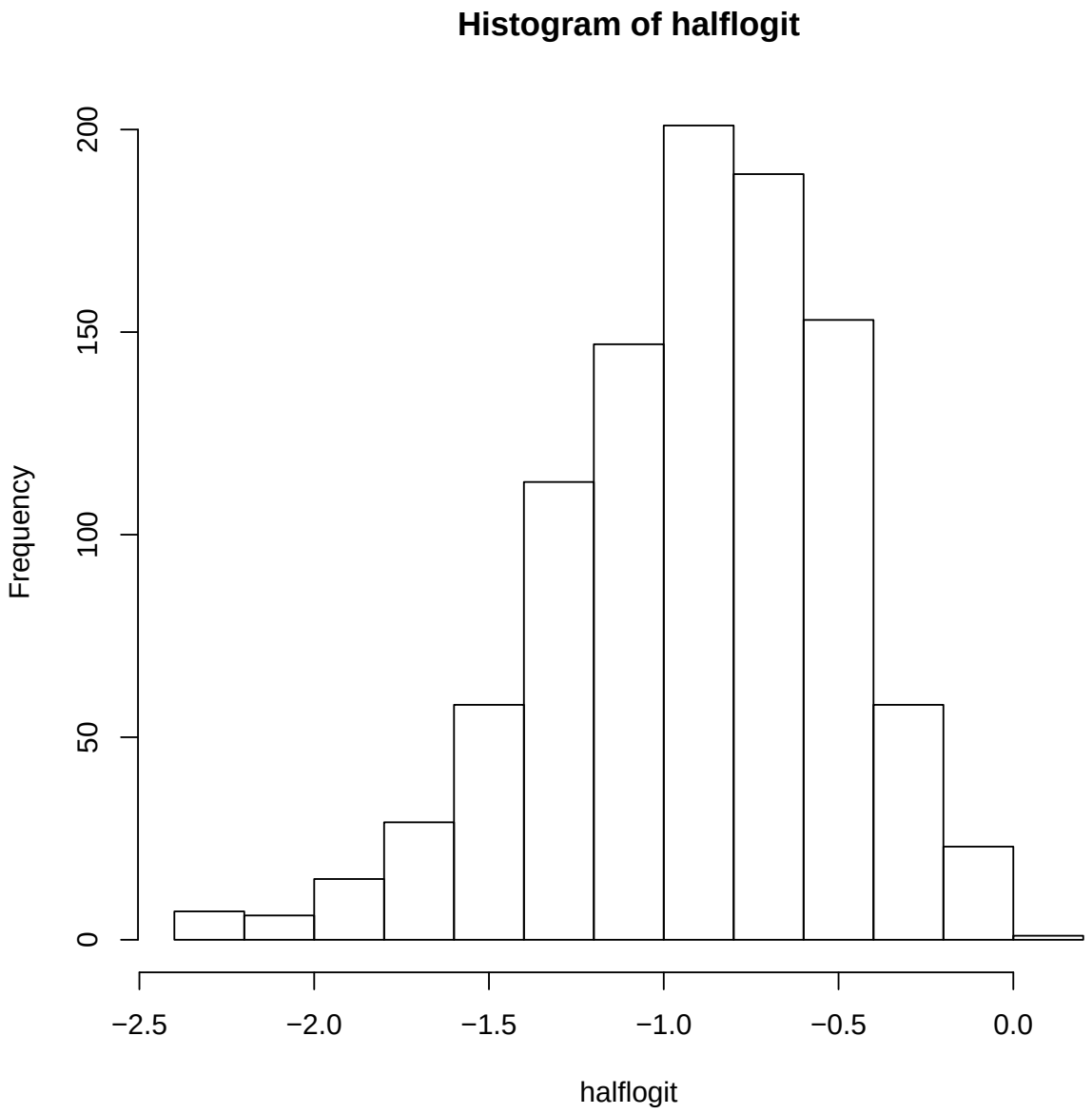

```
halflogit <- log(probs) - (1/2)*(log(probs)+log(1-probs))
hist(halflogit)
```




```
#right skewed distribution (large tail to the left)
probs <- rbeta(1000,2,10)
hist(probs)
```




```
halflogit <- log(probs) - (1/2)*(log(probs)+log(1-probs))
hist(halflogit)
```



In line 2(e) of the boosting algorithm we compute the weight. Let's plug in some numbers and see if we understand the outcomes.

```
# Line 2(d)
line2d <- function(P) (1/2)*log(P/(1-P))
line2e <- function(y,f) exp(-y*f)

# Good prediction
line2e(1,line2d(0.8))

#-# [1] 0.5

line2e(-1,line2d(0.2))
```

```
#-# [1] 0.5

# Bad prediction
line2e(1,line2d(0.2))

#-# [1] 2

line2e(-1,line2d(0.8))

#-# [1] 2

# Conclusion: the weight for a given instance is
# bigger when the instance received a bad prediction
# In other words, instances that are difficult
# to predict will receive more attention in
# subsequent models (i.e., they have a larger
# likelihood of being selected in the bootstrap sample)
```

Figure 2.21 shows the stochastic boosting visually. The process stops when a given number of models has been built.

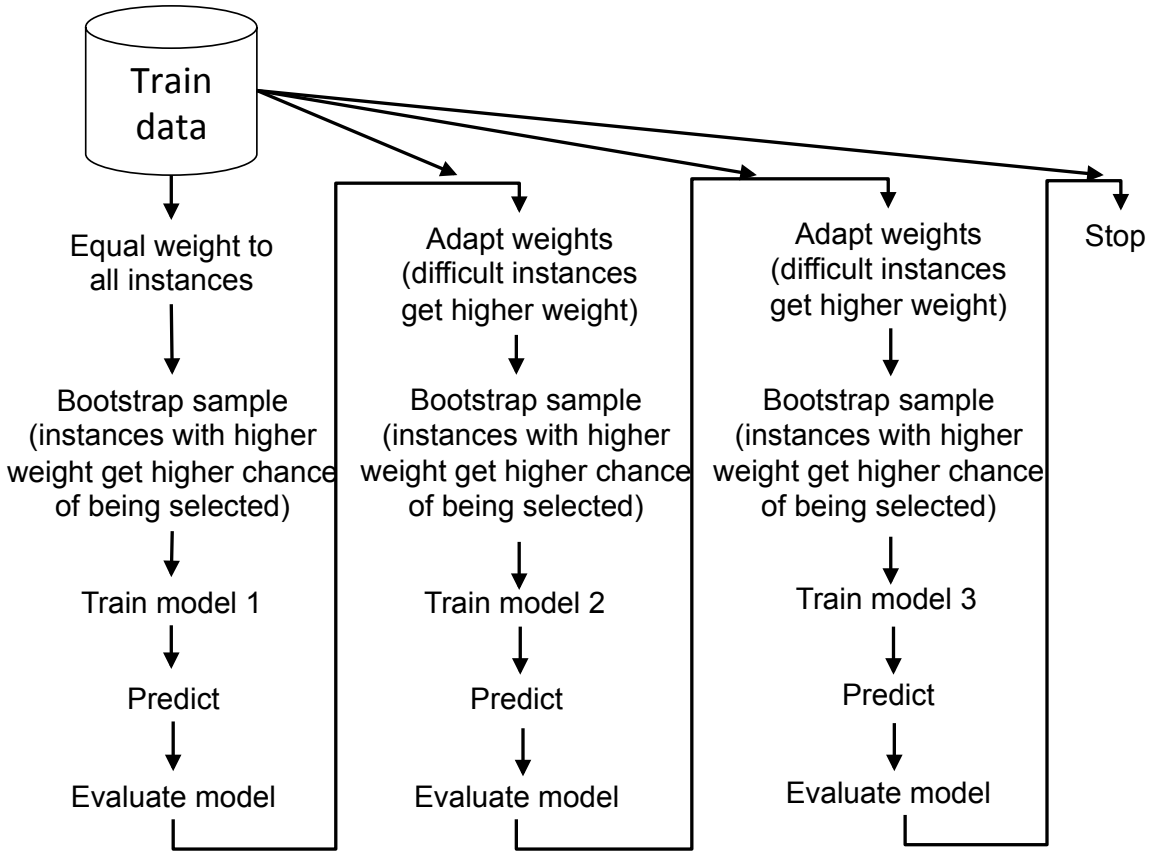
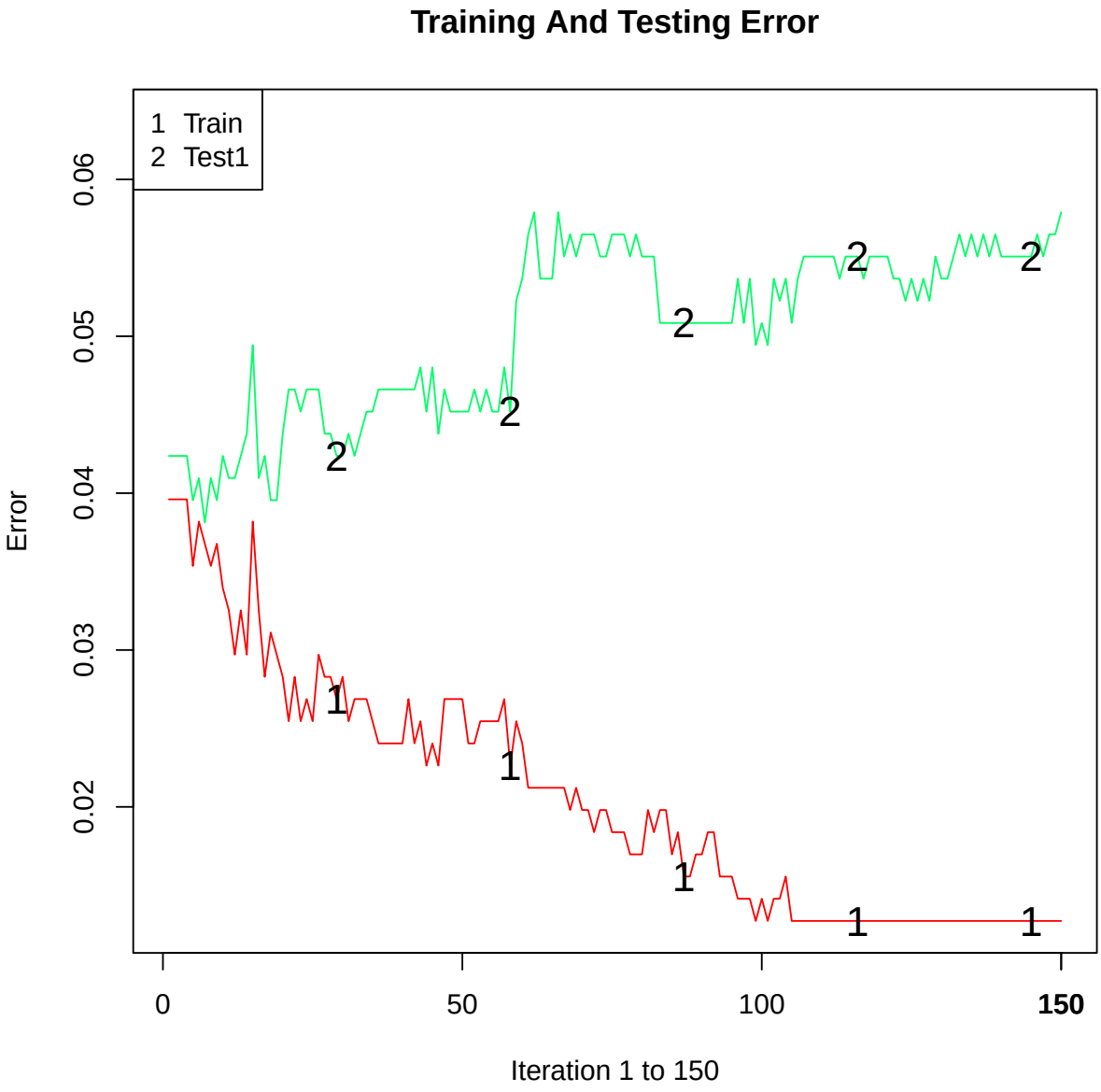


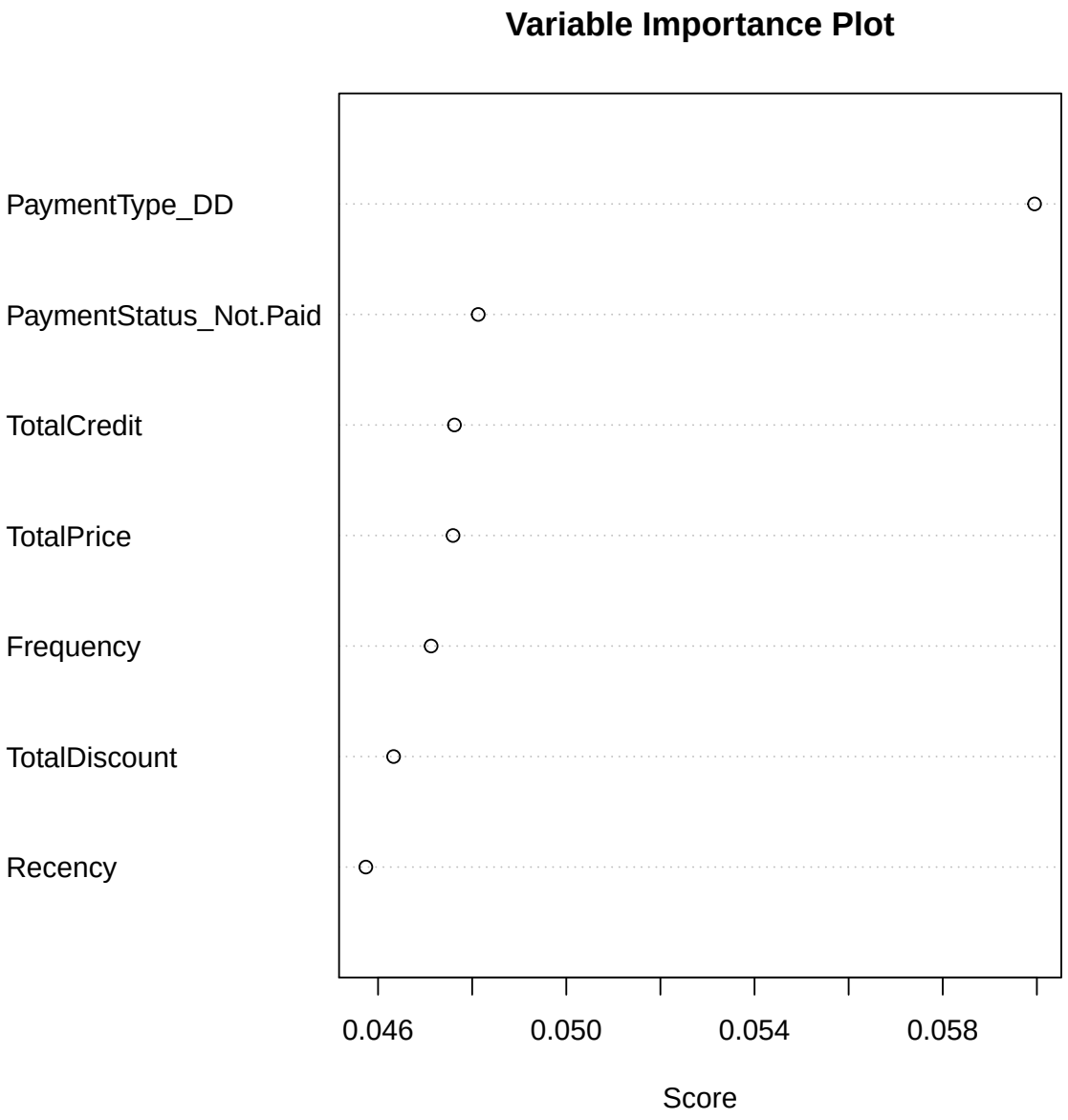
Figure 2.21: Stochastic boosting

The following code chunk shows how to create a boosting model in R. The code is available at: http://ballings.co/hidden/aCRM/code/chapter2/ModelingAB.R.

```
source("http://ballings.co/hidden/aCRM/code/chapter2/read_data_sets.R")

#load the package AUC to evaluate model performance
if(require('AUC')==FALSE)  {
  install.packages('AUC',
                   repos="https://cran.rstudio.com/",
                   quiet=TRUE)
  require('AUC')
}

#load the package ada
if(require('ada')==FALSE)  {
  install.packages('ada',
                   repos="https://cran.rstudio.com/",
                   quiet=TRUE)
  require('ada')
}

?ada
ABmodel <- ada(yTRAINbig ~ . ,
               BasetableTRAINbig,
               iter=150)
predAB <- as.numeric(predict(ABmodel,
                             BasetableTEST,
                             type="probs")[,2])
AUC::auc(roc(predAB,yTEST))

#-# [1] 0.9281331

#plotting learning curve
ABmodel <- ada(x=BasetableTRAIN,
               y=yTRAIN,
               test.x=BasetableVAL,
               test.y=yVAL,
               iter=150)
plot(ABmodel,test=TRUE)
```




```
#variable importances
#Same as in single decision tree but now aggregated across all trees:
#sum of the decrease in impurity for each of the variables at each node
varplot(ABmodel)
#getting them in a character string
varplot(ABmodel,type="scores")
```




```
#-#         PaymentType_DD PaymentStatus_Not.Paid
#-#             0.05995091             0.04812824
#-#            TotalCredit             TotalPrice
#-#             0.04762344             0.04759289
#-#              Frequency          TotalDiscount
#-#             0.04712728             0.04633012
#-#                Recency
#-#             0.04573994

#Tune the number of iterations
ABmodel <- ada(x=BasetableTRAIN,
```

```
              y=yTRAIN,
              iter=which.min(ABmodel$model$errs[,"test.err"]))

predAB <- as.numeric(predict(ABmodel,
                             BasetableTEST,
                             type="probs")[,2])
AUC::auc(roc(predAB,yTEST))

#-# [1] 0.7639053
```

2.5.14 Rotation forest

Breiman (2001) noted the positive effect of principal component analysis (PCA) on the performance of random forest models. One could apply a PCA on the orginal dataset once, and apply a random forest on the principal components instead of on the raw variables. An alternative would be to apply a PCA on each bootstrap sample that is generated for the trees in the forest.

Rodriguez and Kuncheva (2006) further explored the benefits of PCA on ensemble performance and introduced a method called rotation forest. Rotation forest can be summarized as follows. For i=1 ... L models in the forest:

- Prepare a rotation matrix
  - Split the variables randomly in K equally sized disjoint subsets of variables: $F_{i,j}$
  - For j=1 ... K:
    - Select only variables in variable subset $F_{i,j}$
    - Keep only the rows pertaining to one randomly selected class (to promote diversity)
    - Generate bootstrap sample (again to promote diversity)
    - Apply PCA and store coefficients
  - Rearrange columns with coefficients so as to match original dataset and call it rotation matrix
- Multiply rotation matrix with the original data and grow a tree on the resulting dataset

In essence, what is happening here, is that for each tree we are randomnly cutting up the data, estimating PCA coefficients on each piece, arranging those coefficients in a matrix, and using that matrix to project data onto the principal components. A simpler approach would be, for each tree, to build a boostrap sample, estimate a PCA once on the boostrap sample and store the coefficients in a matrix, and use the coefficients to project data onto the principal components. The difference lies in how the coefficient matrix is built. In rotation forest the coefficient matrix is patched together from multiple PCAs estimated on subsets, whereas in the simpler approach the coefficient matrix is estimated as a whole on one set. The underlying reason for taking the more complex path in rotation forest is diversity generation.

In the prediction phase we project our new data on the principal components using the PCA coefficients from the training phase. We then make predictions using our trees, and average the predictions to obtain the ensemble prediction.

The following code chunk shows how to create a rotation forest model in R. The code is available at: `http://ballings.co/hidden/aCRM/code/chapter2/ModelingRoF.R`.

```
source("http://ballings.co/hidden/aCRM/code/chapter2/read_data_sets.R")

#load the package AUC to evaluate model performance
if(require('AUC')==FALSE)  {
  install.packages('AUC',
                   repos="https://cran.rstudio.com/",
                   quiet=TRUE)
  require('AUC')
}

#load the package ada
if(require('rotationForest')==FALSE)  {
  install.packages('rotationForest',
                   repos="https://cran.rstudio.com/",
                   quiet=TRUE)
  require('rotationForest')
}

#-# Loading required package: rotationForest

RoF <- rotationForest(x=BasetableTRAINbig,
                      y=as.factor(yTRAINbig),
                      L=10)

predRoF <- predict(RoF,
                   BasetableTEST)
head(predRoF)

#-# [1] 0.04293022 0.01261777 0.01261777 0.01261777 0.01261777
#-# [6] 0.44781589

AUC::auc(roc(predRoF,yTEST))

#-# [1] 0.9046889
```

2.6 Model Evaluation

2.6.1 Performance measures

2.6.1.1 Binary classification models

Deterministic binary classifiers make a prediction {0,1} of whether or not an event will take place. The performance of these classifiers can be assessed by applying them to an untouched test dataset and compare the predictions with the observed responses {0,1}. Popular performance measures are accuracy, sensitivity and specificity. In contrast to deterministic classifiers, scoring classifiers provide a score [0,1] of an event. When classifiers are calibrated this score can be conceived of as a probability or the classifier's confidence that an event will take place. From that perspective a scoring classifier can be considered superior to a deterministic one, both in use value and performance evaluation, because more information results from its deployment. In the evaluation of both deterministic and scoring classifiers we use a confusion matrix (Figure 2.22).

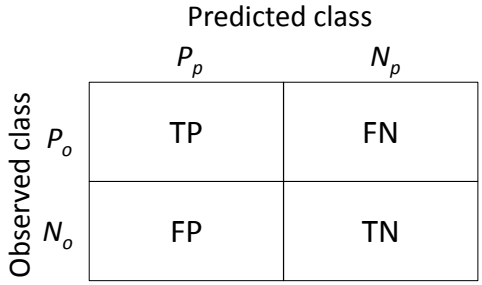


Figure 2.22: Confusion matrix

The P stands for Positive and the N stands for Negative. In the case of deterministic classifiers we don't have any choice as to where to draw the vertical line in Figure 2.22 that separates P_p from N_p. In scoring classifiers we do. The question then becomes: Where should we draw that vertical line (i.e., what score, also called threshold or cutoff, will we select to determine whether we have a P_p or a N_p)? Should we draw the line more to the right (closer to score 0) or to the left (closer to score 1). As we will see later the right answer is not 0.5, nor is it the median or the mean of the scores. The confusion matrix is the basis for computing a whole range of evaluation measures. Table 2.9 provides a summary of the different measures that can be computed from the confusion matrix. There are two columns. The first column is perfect for deterministic classifiers. We do not need to worry about the threshold in that case as the model takes care of that. In the case of a scoring classifier we do need to determine the threshold to compute these measures. We will see that in a minute. The second column is only for scoring classifiers. We would use these measures if we cannot determine the threshold up front. We have been using one such a measure (AUROC, in short: AUC) all along. These measures involve creating a confusion matrix for all possible thresholds. As we will see, the thresholds t for the measures in the second column,

except AUROC, are percentile ranks of the raw scores. The percentile rank is obtained by simply assigning a rank to the raw scores and dividing those ranks by the total number of raw scores.

Table 2.9: Evaluation measures based on the confusion matrix

Single cutoff measures	Cutoff independent measures
$Bal.\ acc. = \frac{1}{2}(\frac{TP}{P_o} + \frac{TN}{N_o})$	$AUROC = \int_0^1 \frac{TP}{P_o} d\frac{FP}{N_o}$
$Specificity = \frac{TN}{N_o}$	$AUSPC = \int_0^1 \frac{TN}{N_o} dt$
$Sensitivity = \frac{TP}{P_o}$	$AUSEC = \int_0^1 \frac{TP}{P_o} dt$
$Accuracy = \frac{TP+TN}{P_o+N_o}$	$AUACC = \int_0^1 \frac{TP+TN}{P_o+N_o} dt$

Why do we need percentile ranks instead of raw probabilities in the second column? Consider the score distributions of two models in Figure 2.23. The first strategy that we can explore to compare both models is to choose the threshold corresponding to a given probability of the event. Let's say we compare both models in terms of sensitivity by using $t = 0.7$.

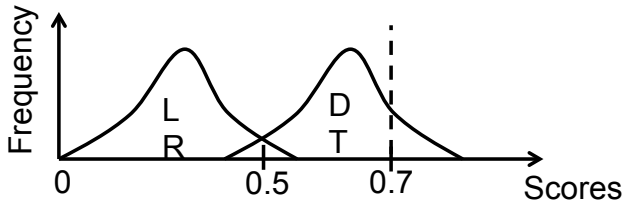


Figure 2.23: Scores of a logistic regression and decision tree model

This is what we would get: sensitivity(t=0.7, LR)=0 and sensitivity(t=0.7, DT) $\geq$ 0. We would conclude that the decision tree outperforms the logistic regression but this is not correct. The problem is that the models are not calibrated. A classifier is well calibrated when the empirical class membership $P(c|p(x) = t)$, with c the observed class, $p(x)$ the predicted probability, and t the threshold value, converges to the $p(x) = t$ when the number of classified instances goes

to infinity (Murphy and Winkler, 1977). More intuitively, if we consider the instances to which a classifier assigns a probability $p(x) = 0.6$, then 60% of these instances should be members of the observed class c (Zadrozny and Elkan, 2002). As Drummond and Holte (2006) and Flach et al. (2011) note, most often classifiers are not calibrated, so this strategy may not always be appropriate. We can calibrate models to alleviate this problem. To create a calibration model, also called a calibrator, we first bin our data based on the scores. Next we compute the proportions of 1s per bin. Finally we train a model based on the resulting dataset to transform scores into probabilities. Consider an example of nine instances (i.e., customers) in Figure 2.24.

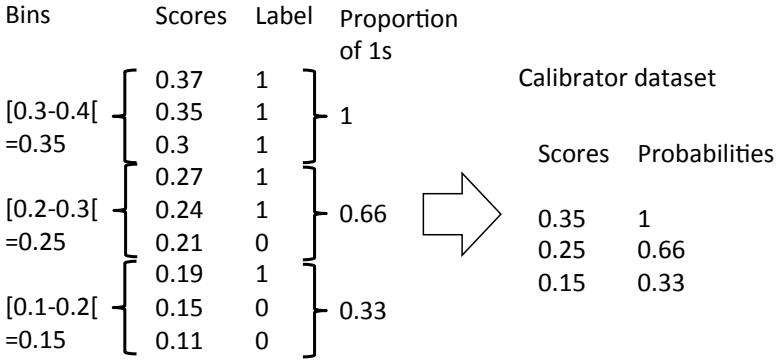


Figure 2.24: Creating a dataset for calibration

The second strategy to choosing a cutoff is used in screening applications such as medical screening, customer targeting, and fraud detection (Hand and Anagnostopoulos, 2013). This strategy consists in choosing the cutoff so that a fixed proportion of the sample has scores above it. For example, the cutoff may be chosen so that 10% of the sample has a higher score and can subsequently be selected (e.g., for treatment or investigation). Consider the following example in Figure 2.25. Suppose we want to target the top 10% of instances based on their event scores. Figure 2.25 then tells us that we need to take $t = 0.5$ for the Logistic Regression and $t = 0.7$ for the Decision Tree. If we then compute sensitivities we get sensitivity(LR,t=0.5) $\geq$ 0 and sensitivity(LR,t=0.5) $\geq$ 0. This means that, in contrast to before, we cannot automatically conclude that the Decision Tree outperforms the Logistic Regression. Now both models have a comparable sensitivity. Indeed, in screening applications it is only the ranking that needs to be preserved and not the height of the probability. If we would do this for all possible cutoffs we obtain a uniform distribution for both algorithms (see Figure 2.26) and the comparison of two models would not be dependent anymore on the distribution of the scores, but solely on whether the instances are better ranked (as it should be).

Hilden and Gerds (2013) devastatingly illustrate that the AUSEC and AUSPC are misleading measures of classifier performance when scores are used. AUSEC and AUSPC integrate over the distribution of all possible values of the threshold *t*, *f(t)*. Since *f(t)* is dependent on the classifier, the way the two measures integrate performance is classifier-specific, while it should be problem or task specific. For example, it would be incorrect to say that it would be likely that we adopt a probability of 0.8 for *t* when a decision tree is used and that it would be likely to choose a probability of 0.6 when a Neural Network is used. To be correct the choice of *t* should be a

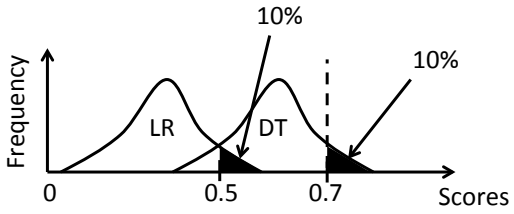


Figure 2.25: Rates

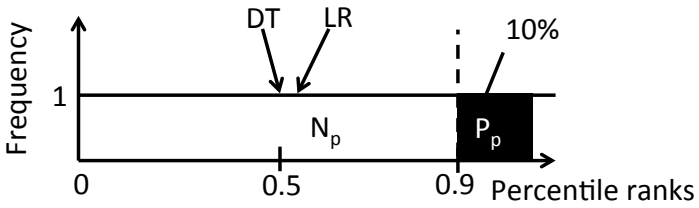


Figure 2.26: Percentile ranks

property of the task or problem and not of the classifier (Hand and Anagnostopoulos, 2013). Letting *f(t)* differ between the classifier to be compared is incoherent and that is the reason why we used percentile ranks: *f(t)* is identical for both models.

Figure 2.27 is another example of how working with rates solves the problem of uncalibrated models. It features two models with identical sensitivity. Suppose we select a cutoff of 0.75. The Decision Tree outperforms the Logistic Regression when we work with raw scores. When we use rates, both are identical as should be.

This strategy is superior to calibrating models because the calibration process itself is based on a model which is turn is only a representation of reality. Working with rates still has a downside though. We need to know, when we are comparing models, what percentage of instances, and hence the threshold, we are about to target when we will deploy our model. However, at the time of model evaluation, the appropriate threshold may be unknown. The specific future circumstances in which the model will be deployed may change. This makes threshold dependent model performance criteria impossible to calculate at the time of the model evaluation phase (Hand, 2005). For example, it might be the case that a company reduces the marketing budget between model creation and deployment. This would effectively reduce the number of customers we can target. A solution to this problem is the use of cutoff independent evaluation criteria (Hand, 2005) (see column 2 in Table 2.9).

There is one other classifier performance measure that we haven't covered yet. It is called the Top Decile Lift and it is defined as follows. This measure is very often used in consultancy.

$$\textit{Top decile lift} = \frac{\frac{P_{Top10\%}}{P_{Top10\%}+N_{Top10\%}}}{\frac{P}{P+N}} \tag{2.39}$$

In that equation all P and N are observed as opposed to predicted. The Top Decile Lift is often accompanied by the lift curve. A lift curve can be created as follows: (1) sort the data

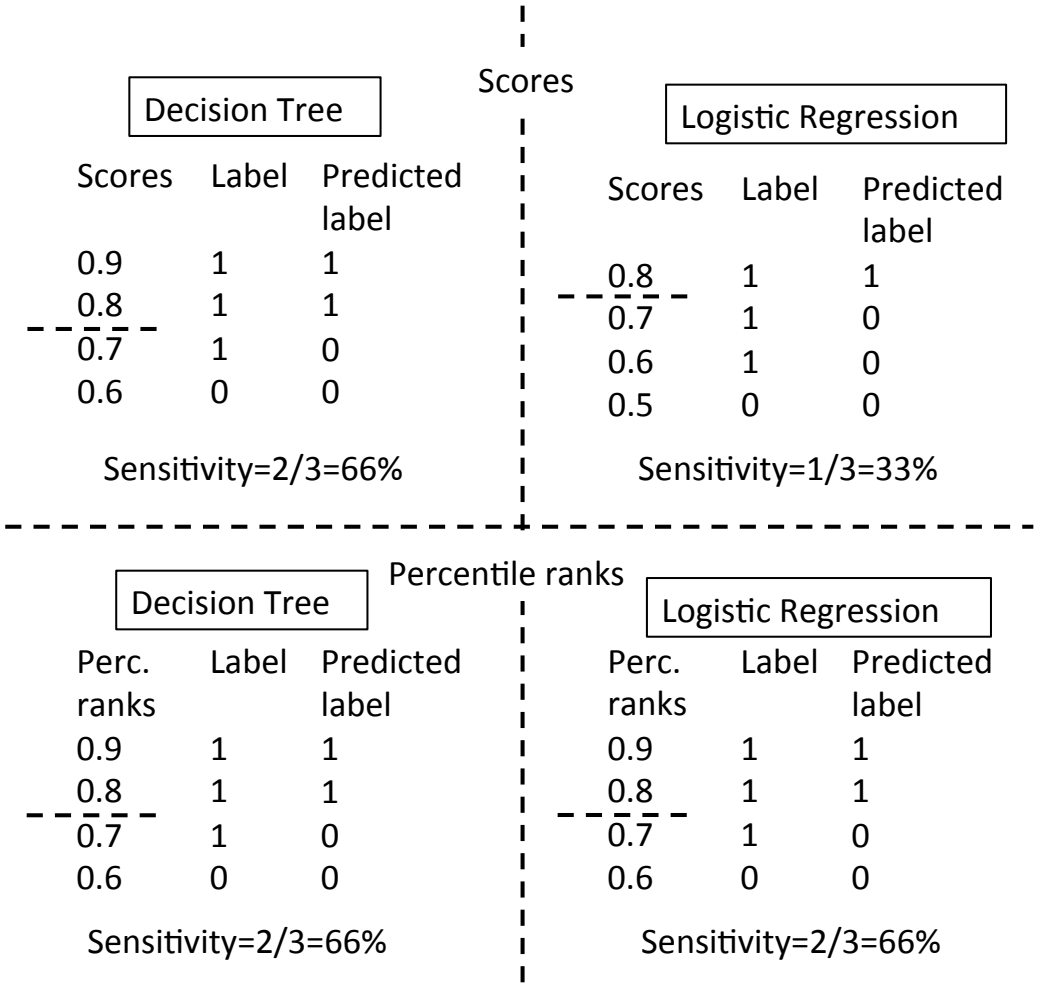


Figure 2.27: Example of using rates

based on the scores from high to low, (2) divide the data in one hundred buckets of equal size (these go on the x-axis), (3) compute the proportion of 1s in each bucket (these go in the y-axis). This results in Figure 2.28. The Top Decile Lift is hence the lift in the top bucket.

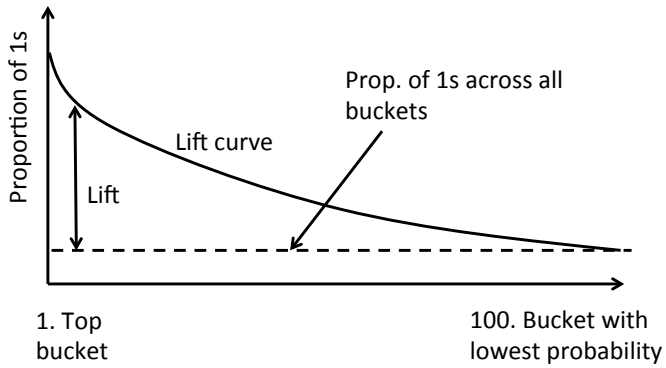
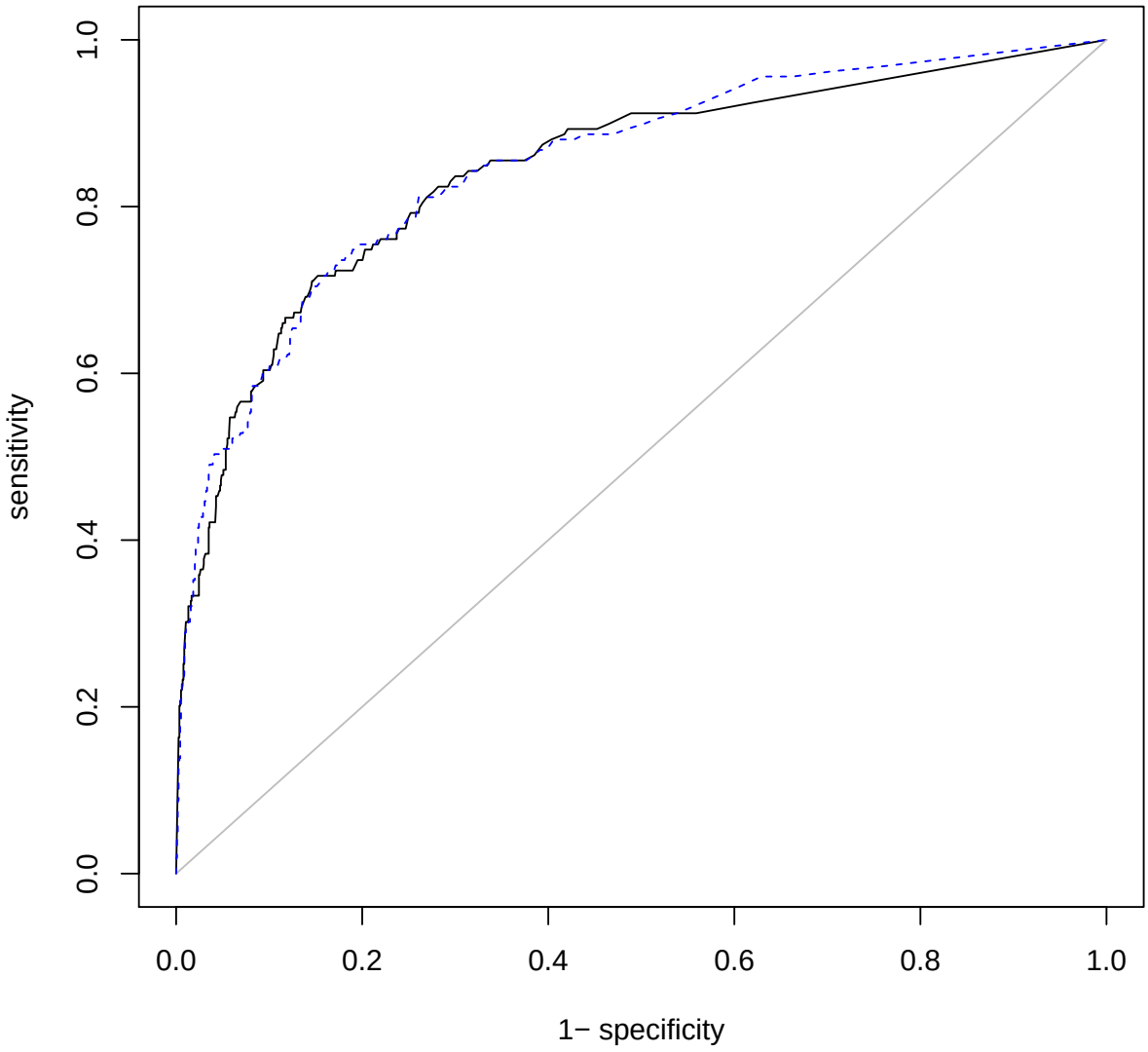


Figure 2.28: Example of a lift curve

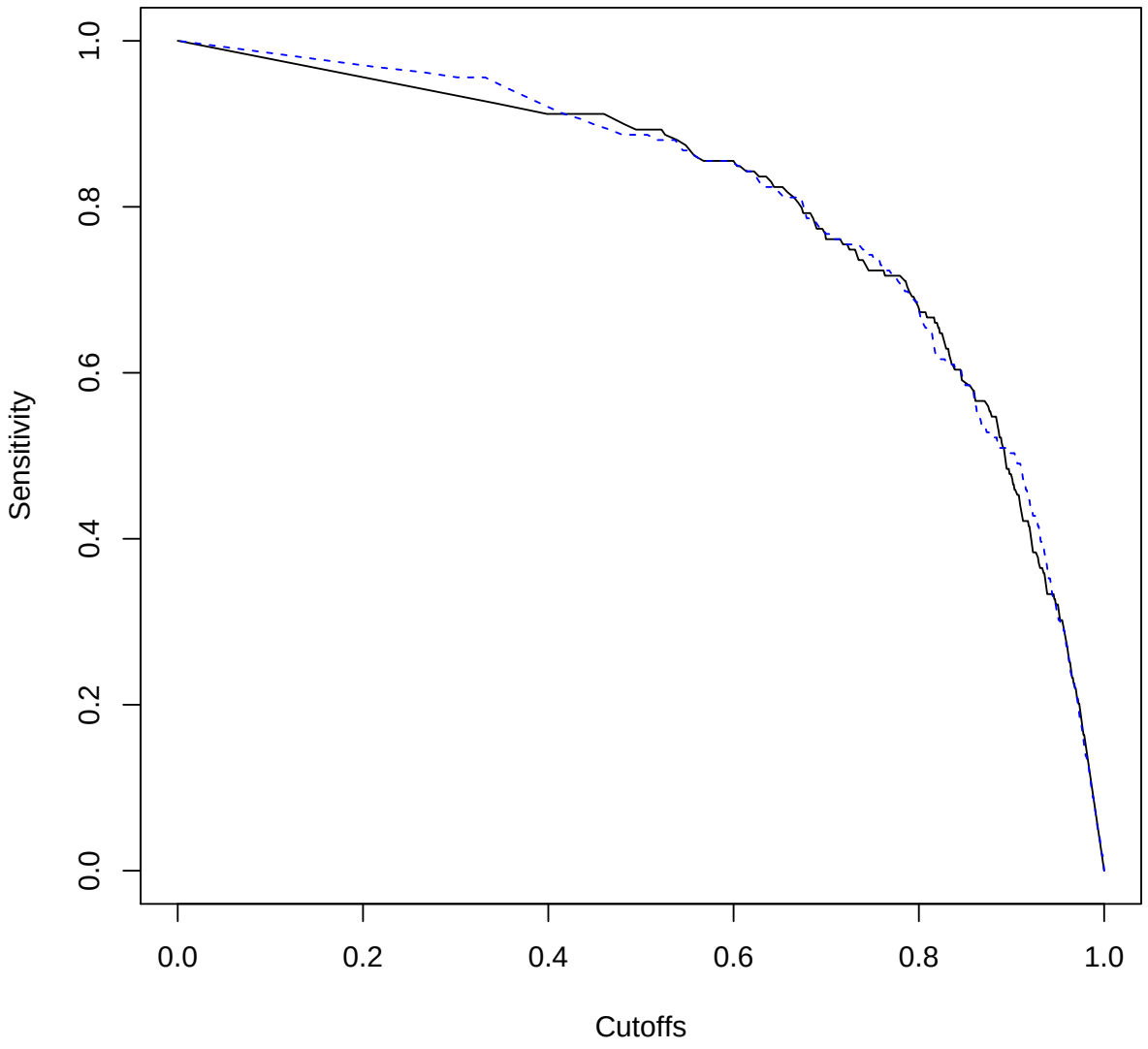
The following code block shows how to compute and plot the AUROC, AUACC, AUSPC, AUSEC, and how to compute the top decile lift.

```
#Load the AUC package
if (!require("AUC")){
    install.packages('AUC',
                     repos="https://cran.rstudio.com/",
                     quiet=TRUE)
    require("AUC")
}

#Load and look at the data
data(churn)
str(churn,vec.len=0.5)

#-# 'data.frame': 1302 obs. of  3 variables:
#-#  $ predictions : num  0 ...
#-#  $ predictions2: num  0 ...
#-#  $ labels      : Factor w/ 2 levels "0","1": 1 ...

###########AUROC
plot(roc(churn$predictions,churn$labels))
plot(roc(churn$predictions2,churn$labels),
     add=TRUE,lty=2,col="blue")
```



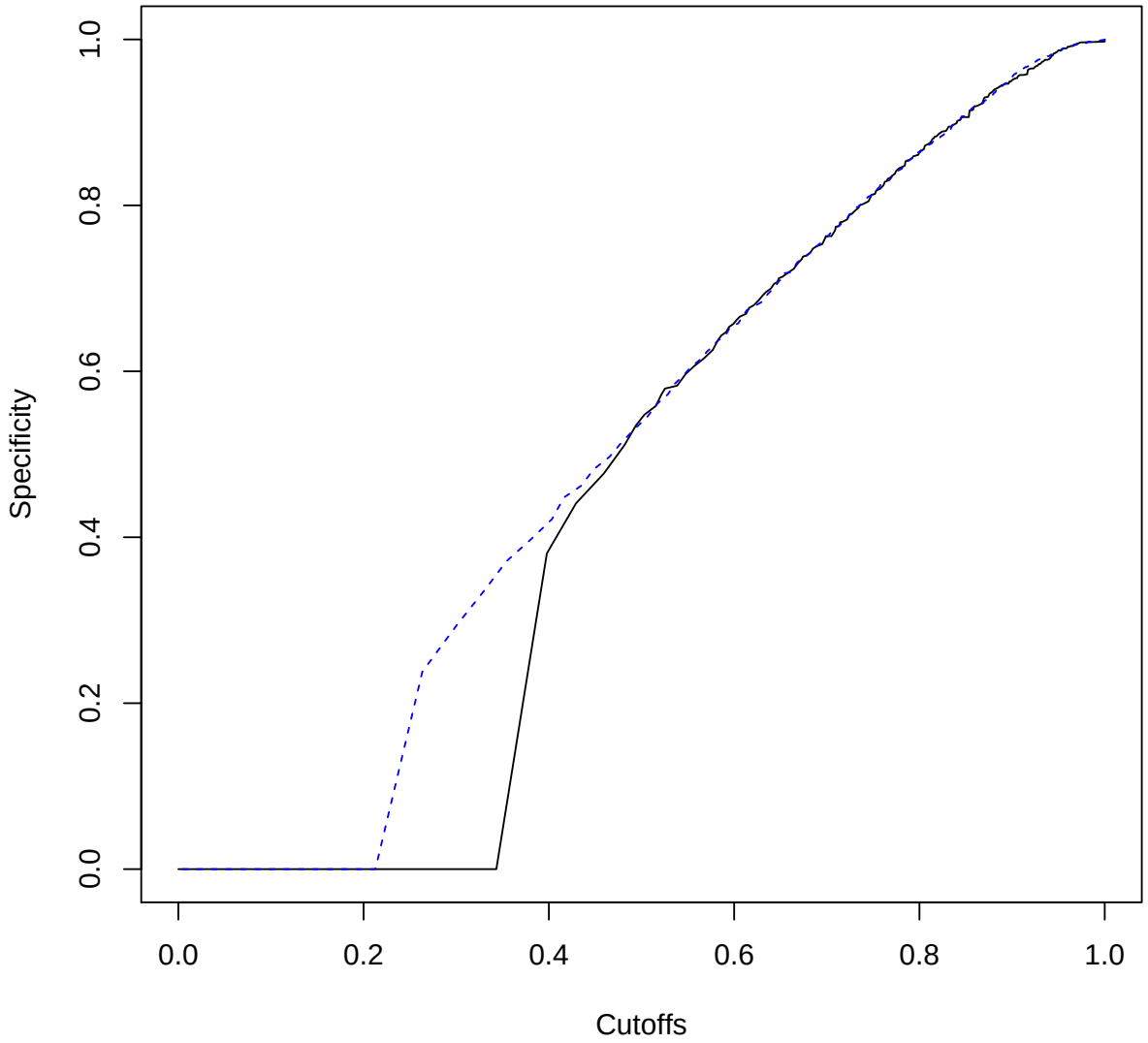

```
# Both predictions are competitive,
# but predictions2 is a little bit stronger in
# higher values for (1-specificity) (i.e.,
# low cutoffs to classify a prediction as 1)
AUC::auc(roc(churn$predictions,churn$labels))

#-# [1] 0.8439201

AUC::auc(roc(churn$predictions2,churn$labels))

#-# [1] 0.8491529
```

```
###########AUSEC
plot(sensitivity(churn$predictions,churn$labels))
plot(sensitivity(churn$predictions2,churn$labels),
    add=TRUE,lty=2,col="blue")
```

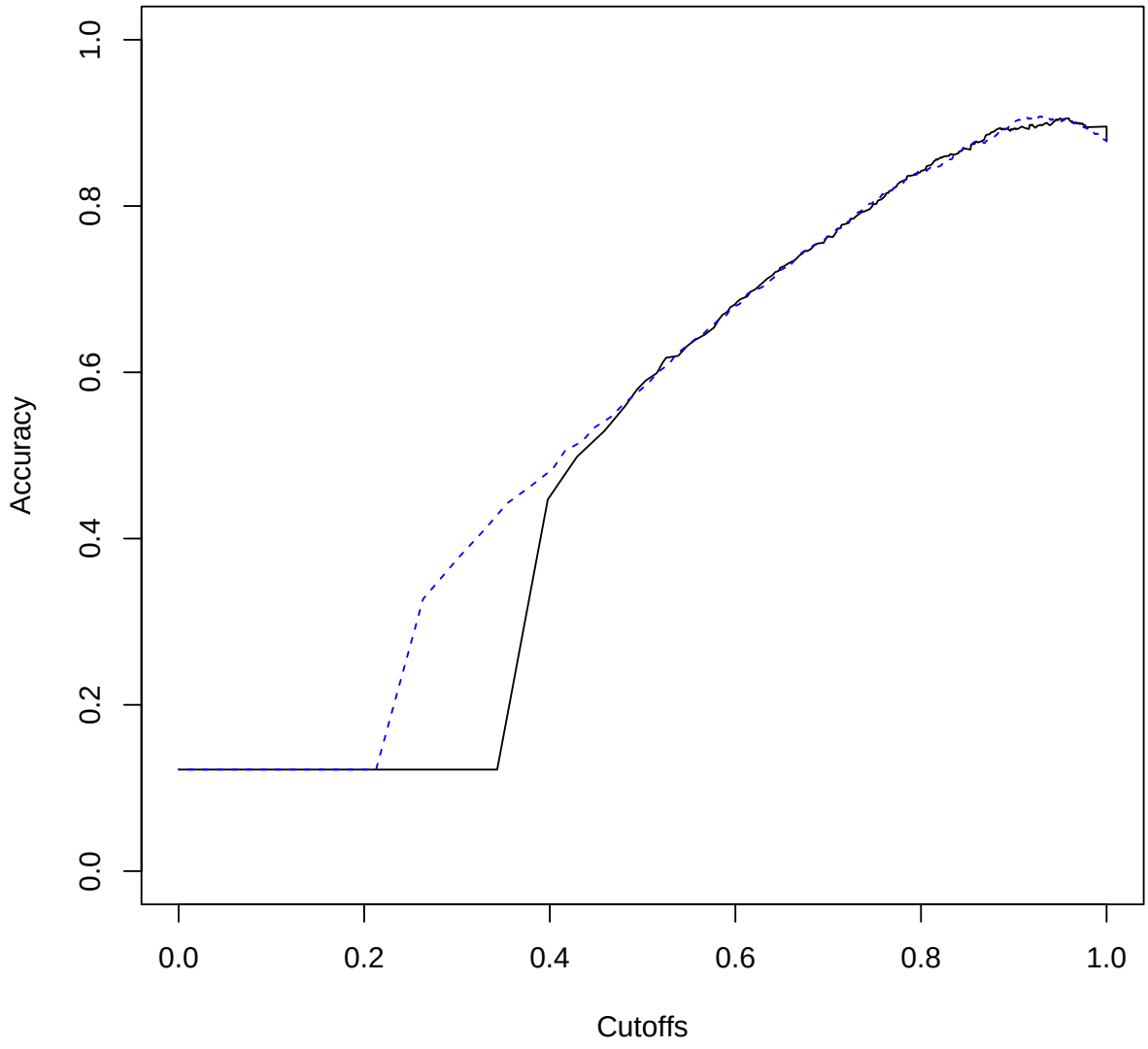



```
# Both predictions are competitive,
# but predictions2 is a little bit stronger in
# lower cutoffs.
AUC::auc(sensitivity(churn$predictions,churn$labels))

#-# [1] 0.8026259
```

```
AUC::auc(sensitivity(churn$predictions2,churn$labels))

#-# [1] 0.8072752

###########AUSPC
plot(specificity(churn$predictions,churn$labels))
plot(specificity(churn$predictions2,churn$labels),
     add=TRUE,lty=2,col="blue")
```

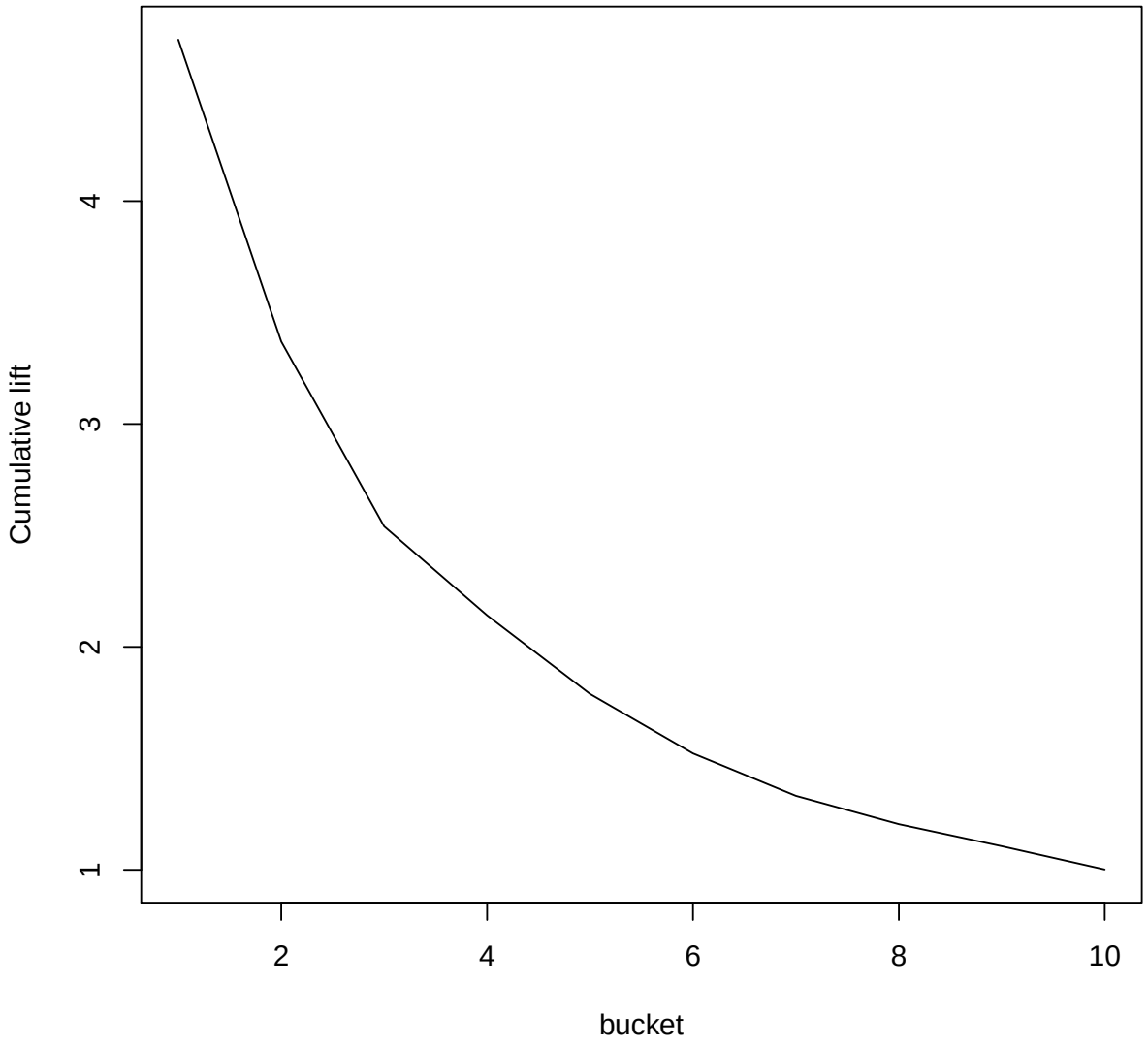


```
AUC::auc(specificity(churn$predictions,churn$labels))

#-# [1] 0.4591936

AUC::auc(specificity(churn$predictions2,churn$labels))

#-# [1] 0.5013036

###########AUACC
plot(accuracy(churn$predictions,churn$labels))
plot(accuracy(churn$predictions2,churn$labels),
     add=TRUE,lty=2,col="blue")
```




```
# predictions2 is definitely stronger in the
# lower cutoff region.
AUC::auc(accuracy(churn$predictions,churn$labels))

#-# [1] 0.5034279

AUC::auc(accuracy(churn$predictions2,churn$labels))

#-# [1] 0.5395265

############Top Decile Lift
```

```
if (!require("lift")){
    install.packages('lift',
                     repos="https://cran.rstudio.com/",
                     quiet=TRUE)
    require("lift")
}

#-# Loading required package: lift

TopDecileLift(churn$predictions,churn$labels)

#-# [1] 4.724

# This means that the model identifies positives 4.724
# times more than simply selecting instances at random

plotLift(churn$predictions,churn$labels)
```



2.6.1.2 Multiclass classification models

2.6.1.3 Regression models

One of the most often used performance measures of predictive regression models is called the root mean square error of prediction (RMSEP) (Cederkvist et al., 2005), defined as follows:

$$RMSEP = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{2.40}$$

where *n* is the number of objects in the test sample, the y_i's are the responses for the test observations and the $\hat{y}_i$'s are the predicted responses of the test observations (Cederkvist et al., 2005).

RMSEP works well when comparing models. However, our favorite measure is the percentage of variance explained, where $\bar{y}$ is the mean of the response variable, computed as:

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} \tag{2.41}$$

2.6.2 Cross- Validation

Instead of building and testing a model once on one particular training-testing split, we can create a model multiple times based on different training and testing sets. We would create these sets based on different random seeds. This reduces the chance of getting an optimistic or pessimistic reading of model performance. To get the final model performance we simply average the performances on the different test sets. There are a multitude of cross- validation schemes but here we cover only the two most common ones: ten fold cross-validation (10fcv) and five times twofold cross-validation (5x2fcv).

In 10fcv we cut the data into ten subsets, use nine of them to train our model and one of them to test our model. We do this ten times, each time with a different subset as testing set. The process is displayed in Figure 2.29. The disadvantage of 10fcv is that 80% of the training sets overlaps across folds. This means that the models will be very similar and hence the variance in the ten performances can be underestimated.

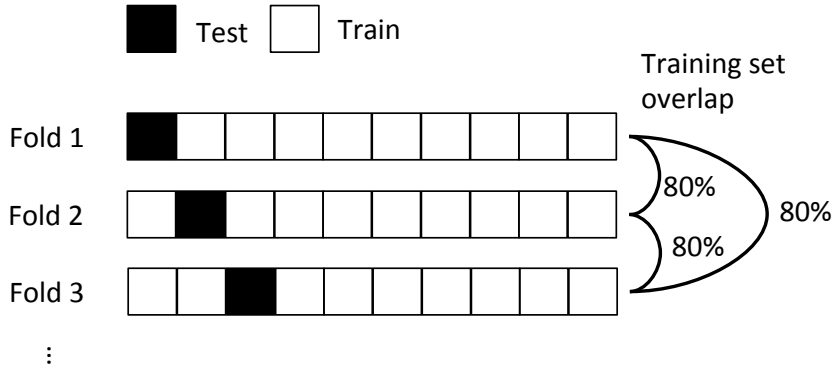


Figure 2.29: 10fcv

This problem is reduced in 5x2fcv. Five times two-fold cross validation randomly partitions the sample in two parts of equal size and repeats this process five times. Each time the first partition is used as a training sample and the second as test sample, and vice versa. This process results in 10 performance measures per model (Dietterich, 1998). The process is displayed in Figure 2.30. In 5x2fcv there is a training overlap of 0%-60%. Therefore models can be less similar and variance can be less underestimated.

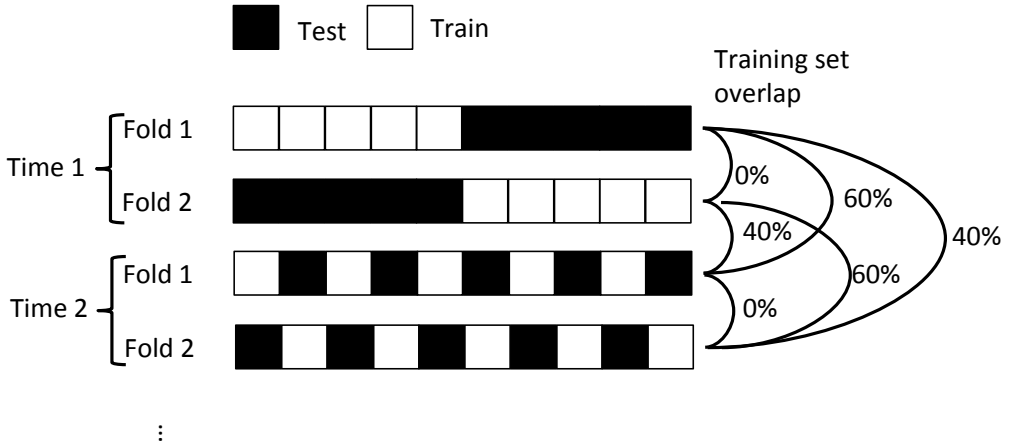


Figure 2.30: 5x2fcv

This is however, only part of the story. In 10fcv the overlap in the test sets is 0% whereas in 5x2fcv the overlap in test sets is 0%-60%. This offsets the claims that 10fcv has less diverse performances on the folds and hence that variance is underestimated. Accordingly, the more diverse training sets in 5x2fcv will be accompanied by testing sets of equal diversity (but less diverse than in 10fcv), and hence variance will be more underestimated than we initially thought. The choice of which type of cross-validation to use is hence not clear cut, and in reality both are equivalent and very much accepted in the academic community.

What do we do next? After cross-validation we have 10 performance values (e.g., 10 AUCs). We need to summarize those values. The question then becomes: which summary measures should we use? We should look at location (central value) and dispersion (variability).

Higher location is better. We could take for example the mean. However, since we only have ten values, we often don't have a normal distribution. Ironically, the Kolmogorov-Smirnov test and similar tests for testing normality of distributions have little power on small samples. They are unlikely to detect abnormalities. In sum, we need a normal distribution to compute the mean, but small samples prohibit us from checking the distribution shape. Therefore we use the nonparametric version of the mean: the median.

Lower dispersion is better. There are two key dispersion components: (1) how spread out are the values near the center and (2) how spread out are the tails? A good dispersion measure captures both components. In what follows we will have a look at our options:

- Range= max - min. This is not a good measure because the spread near the center is not captured. Moreover, it is affected by extreme observations.

- Interquartile range= 75^{th} quantile - 25^{th} quantile. This is not a good measure because the spread near the tails is not measured. It is however less affected by extreme observations.

- Standard deviation ($SD = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(y_i - \bar{y})^2}$) and Variance ($SD^2$). The spread is measured in the center and the tails. However, these measures use the mean (which is the

wrong location is we have non-normal distributions). These measures are also affected by extremes because they use the square difference with the mean.

- Mean Absolute Deviation ($MeanAD = \frac{1}{N}\sum_{i=1}^{N}|y_i - \bar{y}|$). This measures the tail and the center. It does not square the distance from the mean and hence it is less affected by extreme observations. It does however, still use the incorrect location if the distribution is non-normal.

- Median Absolute Deviation ($MedianAD = median|y_i - \tilde{y}|$, with $\sim$ the median). The tail and center spread are measured. It does not use the mean and it is not affected by extremes. This is the best dispersion measure.

In order to determine whether two models are significantly different we follow Demšar's (2006) recommendation to use the Wilcoxon signed-ranks test (Wilcoxon, 1945). The Wilcoxon signed-ranks test (Wilcoxon, 1945) is a non-parametric test, that ranks the differences in performance of two models, while ignoring the signs. Ranks are assigned from low to high absolute differences, and equal performances get the average rank. The ranks of both the positive and negative differences are summed and the minimum of those two is compared to a table of critical values (Demšar, 2006). This test is safer than parametric tests such as the t-test because it does not assume normal distributions or homogeneity of variance (Demšar, 2006) and is less sensitive to outliers. When the assumptions of the t-test are met, the Wilcoxon signed ranks test has less power than the t-test. However, since in this case the sample size is only ten differences, verifying normality and homogeneity of variance is problematic and hence the Wilcoxon signed ranks test is strongly recommended over the paired t-test (Demšar, 2006).

Table 2.10 contains an example of the Wilcoxon signed ranks test for comparing two 5x2fcv cross-validated models. Ties are averaged. First we compute the sum of the ranks of the positive difference: $R^+ = 1.5 + 4 + 3 + 10 + 1.5 + 8 + 6 = 34$. The sum of the ranks of the negative differences are: $R^- = 9 + 6 + 6 = 21$. The min(R^+,R^-)=21. We compare this number with the number in the table of exact critical values for Wilcoxon's test (Table 2.11) for $\alpha = 0.05$, and N=10. If our number is equal to or smaller than the number from the table, then our two models are significantly different. In this case the value is 8, so this means that our models are not significantly different.

Table 2.10: Example of the Wilcoxon signed-ranks test

Time	Fold	AUC model 1	AUC model 2	Difference	Rank absolute difference (lowest=1)
1	1	0.80	0.81	+0.01	1.5 (1)
1	2	0.75	0.78	+0.03	4
2	1	0.80	0.72	-0.08	9
2	2	0.78	0.80	+0.02	3
3	1	0.70	0.80	+0.1	10
3	2	0.75	0.76	+0.01	1.5 (2)
4	1	0.81	0.88	+0.07	8
4	2	0.76	0.72	-0.04	6 (5)
5	1	0.75	0.79	+0.04	6 (6)
5	2	0.79	0.75	-0.04	6 (7)

Table 2.11: Table of critical values for the Wilcoxon test (two tailed significance levels)

N	$p = 0.05$	$p = 0.02$	$p = 0.01$
6	0	-	-
7	2	0	-
8	3	1	0
9	5	3	1
10	8	5	3
11	10	7	5
12	13	9	7
13	17	12	9
14	21	15	12
15	25	19	15

2.6.3 Understanding classifier performance

Prediction error consists of (1) bias, (2) variance, and (3) irreducible error (noise). Noise is of lesser concern because we cannot change anything about it. Therefore we will focus on bias and variance. Figure 2.31 gives an initial impression of what those two terms mean. Consider a single point, called the true point $\tilde{y}$ (not to be confused with the median), denoted by $*$. Consider multiple predictions by different models $\hat{y}_i$, with i={1,2,..., K models} each denoted by a $\bullet$. The average of all the predictions $\hat{y}_i$, is called $\bar{y}$ and is denoted by o. If we would repeat the model building process (e.g., using new data, random initialization) the bias denotes how far off the average of the predictions of the different models is from the true value. The variance then denotes how much predictions of the different models vary. Figure 2.32 further denotes the difference between bias and variance.

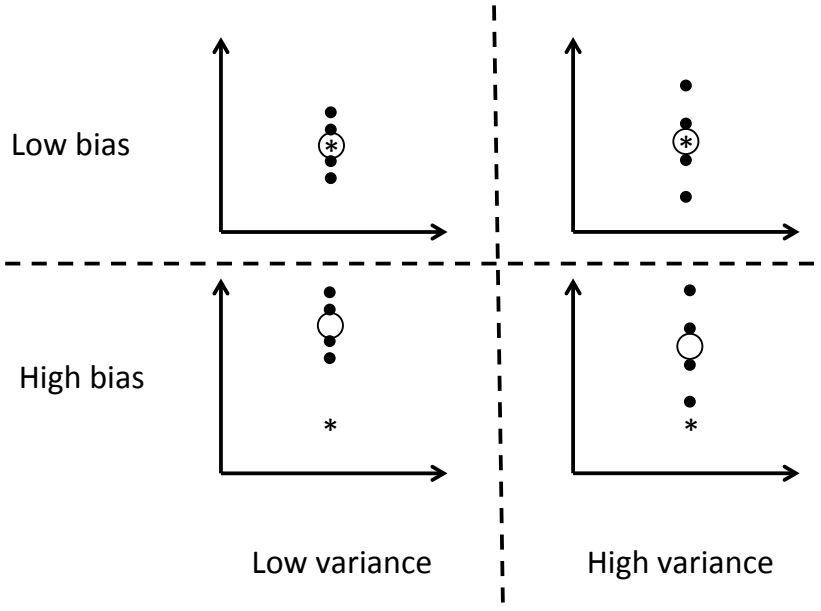


Figure 2.31: Graphical presentation of bias and variance for a single point

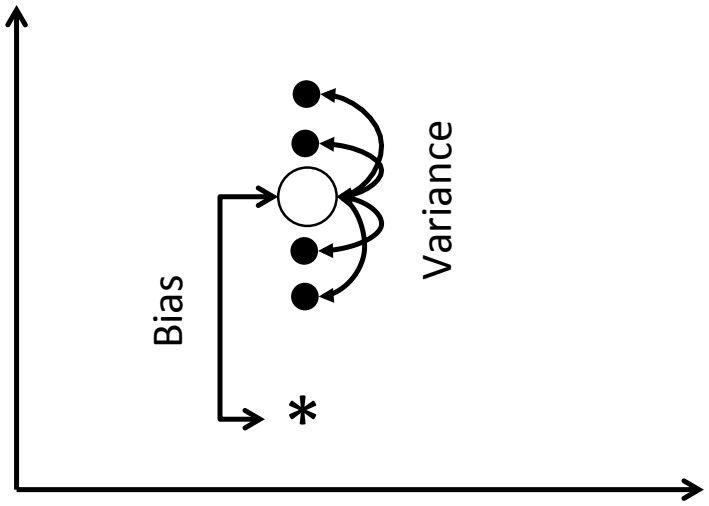


Figure 2.32: Explicit graphical presentation of bias and variance for a single point

In what follows we'll see how to investigate bias and variance mathematically (called bias-variance decomposition). First consider the following notation:

- K= number of predictions
- N= number of instances
- ξ= error of the model
- y= observed response
- y= $\tilde{y} + \epsilon$, with $\tilde{y}$=true function, and ϵ= error term (noise)
- $\hat{y}$= predicted response
- $\bar{y}$= average predicted response across different realizations of the model

The ξ of a model is denoted by:

$$\xi = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

Using the following formula we can decompose ξ into bias, variance and noise. The first term is the squared bias, the second term is the variance and the third term is the squared noise. Take a moment to relate his equation to Figure 2.32.

$$\xi = \sum_{i=1}^{N} \left((\bar{y}_i - \tilde{y}_i)^2 + \frac{1}{K-1} \sum_{j=1}^{K} (\hat{y}_{i,j} - \bar{y}_i)^2 + (y_i - \tilde{y}_i)^2 \right)$$

How do we use this equation in a practical way? First of all, we cannot do anything about noise, but we can do something about bias and variance. Hence we are only interested in the relative magnitudes of bias and variance and we simply set ϵ (noise) to 0. Recall that y= $\tilde{y} + \epsilon$, so y now equals $\tilde{y}$. This helps us simplify our formula to:

$$\xi = \sum_{i=1}^{N} \left((\bar{y}_i - y_i)^2 + \frac{1}{K-1} \sum_{j=1}^{K} (\hat{y}_{i,j} - \bar{y}_i)^2 \right)$$

To compute that equation we need to have multiple models. However, in practice we only have one. The solution is simple: bootstrap your data multiple times, and build a model on each bootstrap sample. Here are the steps in computing the equation:

1. Make K bootstrap samples (keep some data out-of-bag, i.e., set some data aside, e.g., 20%)

2. Learn a model on each bootstrap sample

3. For each model predict $\hat{y}$ on out-of-bag data. We now have, for each instance, one y and $\hat{y}_1, \ldots, \hat{y}_K$.

4. Compute $\bar{y} = \frac{1}{K} \sum_{j=1}^{K} \hat{y}_i$

5. Compute $bias^2 = \sum_{i=1}^{N} (\bar{y} - y_i)^2$

6. Compute variance= $\sum_{i=1}^{N} \frac{1}{K-1} \sum_{j=1}^{K} (\hat{y}_{i,j} - \bar{y}_i)^2$

7. Compare $bias^2$ with variance. If $bias^2 > variance$ then you have a bias problem, otherwise you have a variance problem. We will see how we can use this practically. For now we focus on understanding the nature of bias and variance.

We have set ϵ to 0. If we have multiple data points with the same x-values then we can estimate ϵ. We could also estimate ϵ by pooling y from nearby x-values. But as I mentioned before, we are not really interested in ϵ since we cannot do anything about it.

Up to now the graphical representations of bias and variance were for a single point. It is valuable to see them for multiple points. Figure 2.33 presents bias, Figure 2.34 presents variance, and Figure 2.35 denotes noise.

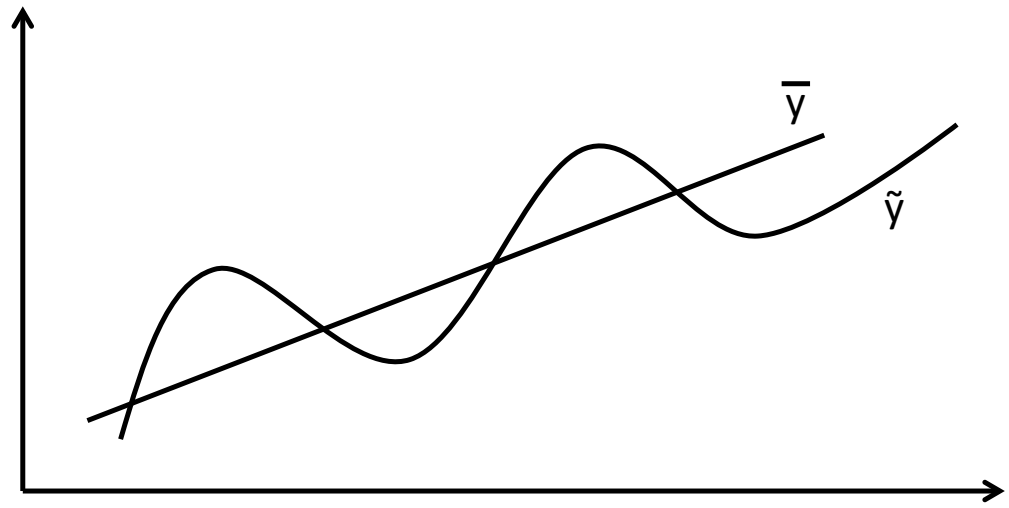


Figure 2.33: Bias

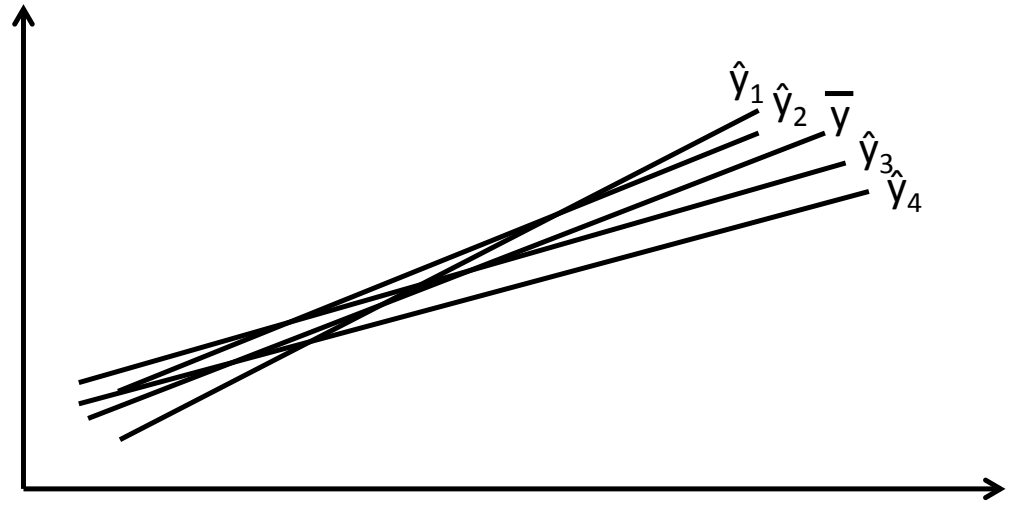


Figure 2.34: Variance

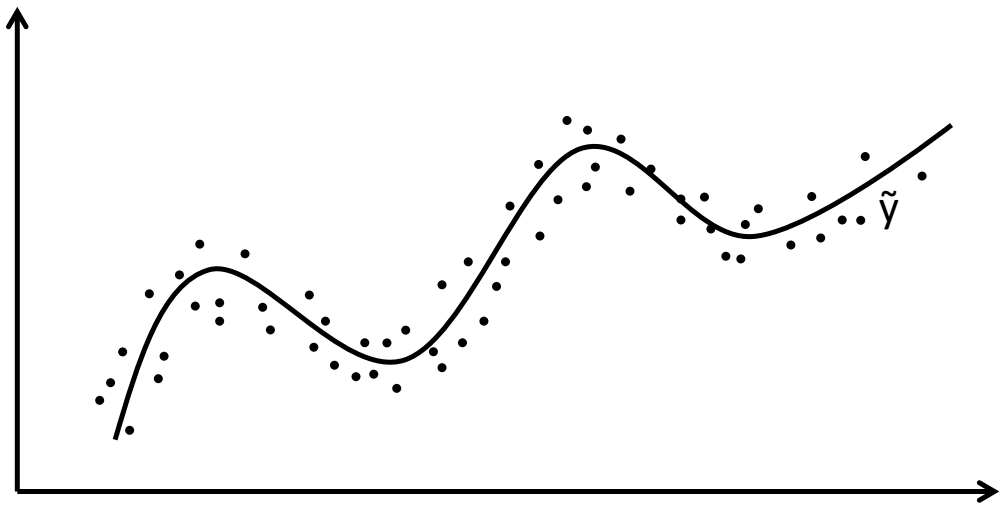


Figure 2.35: Noise

At this point we have described bias and variance in words, graphically and mathematically. Now how can we use this practically? The problem with bias and variance is that if you decrease the former, you will increase the latter and vice versa. Hence the name "bias- variance tradeoff". This is depicted in Figure 2.36. On the x-axis we have model complexity. Examples are the number of parameters, the degree of nonlinearity, and flexibility. If a model, for example a logistic regression model or a decision tree, becomes more complex then bias will decrease. At the same time variance will increase. This is problematic because as we have learned, error is the sum of those components. If a model becomes too complex we call it overfitting. It means that we have low bias and high variance. If it is not complex enough we call it underfitting. It means that we have high bias and low variance. The goal is to balance bias and variance at an optimal complexity. That is where the total error is at its minimum.

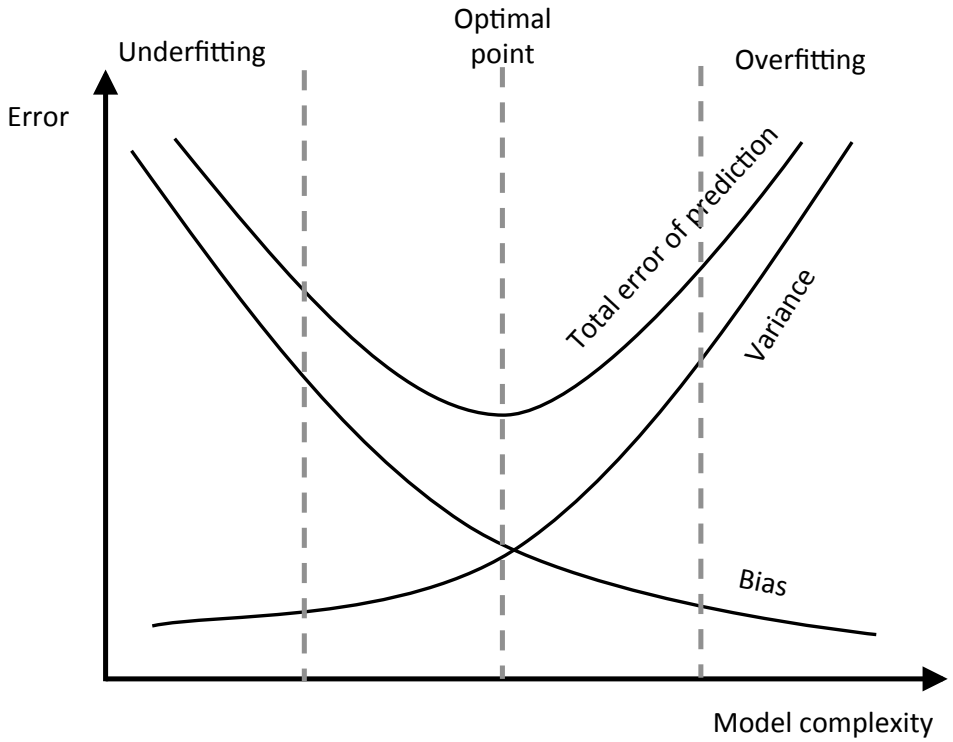


Figure 2.36: The bias- variance tradeoff

How can we diagnose high bias and high variance? We could plot the total error for a range of complexities, just as in Figure 2.36. The problem with this is that, depending on how many points you want on your curve, this is very inefficient. The alternative is to simply compare the training and the test error. This is depicted in Figure 2.37. If the training error and the test error are very similar then there is a bias problem. The solution is to increase complexity. If the training error and the prediction error are very different we have a variance problem. The solution is to decrease complexity.

The culprit of poor model performance is most often overfitting (i.e., high variance). The reason is simply that we live in a time when data is abundantly available. That is also the reason why data mining came into existence. A very elegant solution to overfitting is averaging in the form of ensembles (e.g., bagging, Random Forests, AdaBoost). Consider Figure 2.38. There are two models created by algorithm A and two by algorithm B. Both algorithms have high variance. In addition the algorithm A models have high bias, and ensembling them does not improve performance (i.e., we do not get closer to $*$). On the other hand, the algorithm B models are not biased and ensembling does improve performance (i.e., we do get closer to $*$). We can conclude that ensembling can reduce variance without increasing bias. It is however, only helpful if bias is low. This process was invented by Breiman (1996) and was a major breakthrough in the machine learning community.

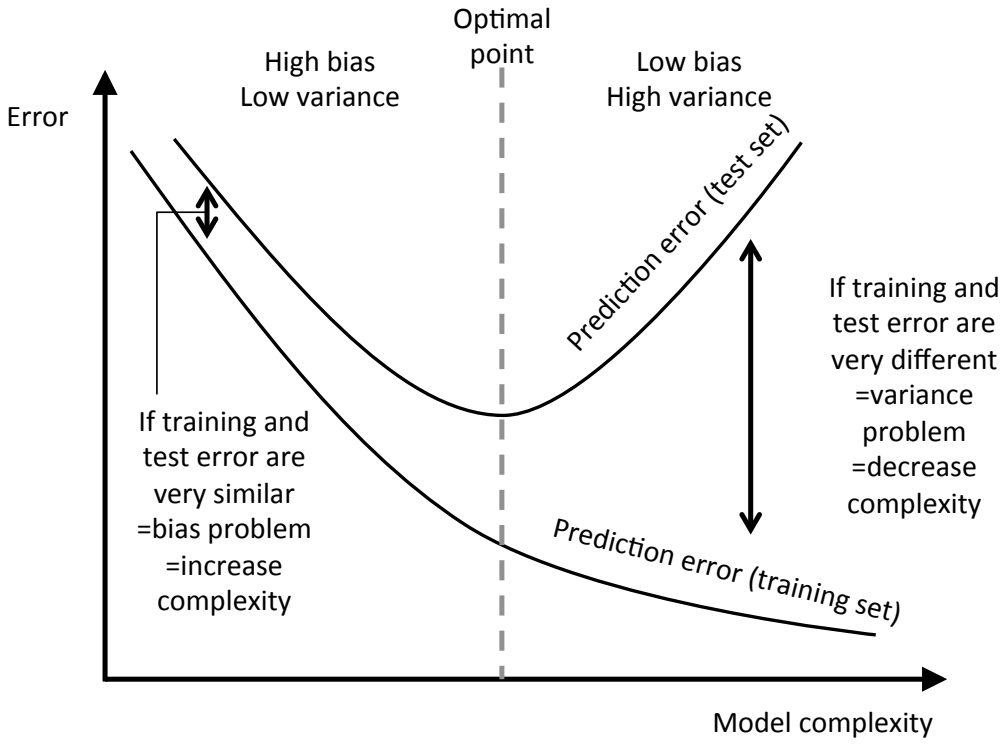


Figure 2.37: Diagnosing bias and variance problems

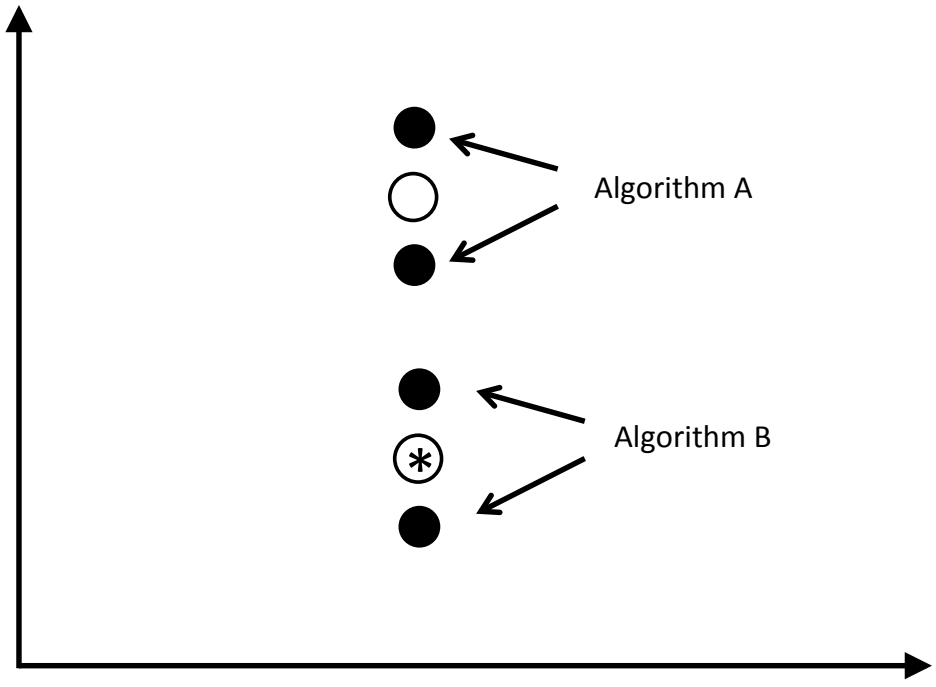


Figure 2.38: Ensembles as a solution to high variance

Figure 2.39 provides an overview of how ensembles impact bias and variance, independently of the complexity of their base classifiers. Base classifiers are the individual team members in the ensembles (the solid lines represent a single team member). Bagging, Random Forest and Boosting reduce variance. Boosting also reduces bias, because it iteratively improves by focuses on the difficult instances.

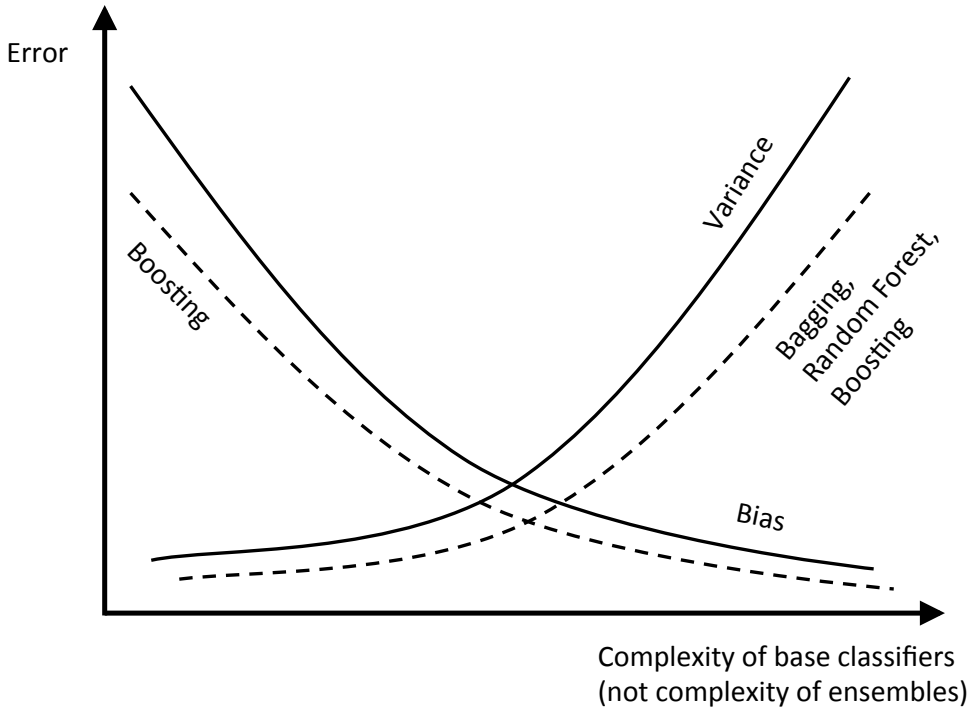


Figure 2.39: The benefits of ensembles

One could conclude then that boosting is always the best choice. Figure 2.40 shows that, that is not necessarily true. On the x-axis we now have ensemble complexity (e.g., number of trees). Because boosting iteratively improves by giving more weight to the instances that are classified incorrectly in the previous iteration, it is also more prone to overfitting. In other words, boosting initially decreases bias, but with an increasing number of iterations it can increase variance. This problem is absent in Random Forest and Bagging. This means that boosting is more sensitive to tuning. Finally, boosting also requires a validation set. And therefore its performance also depends on the representativeness of that validation set.

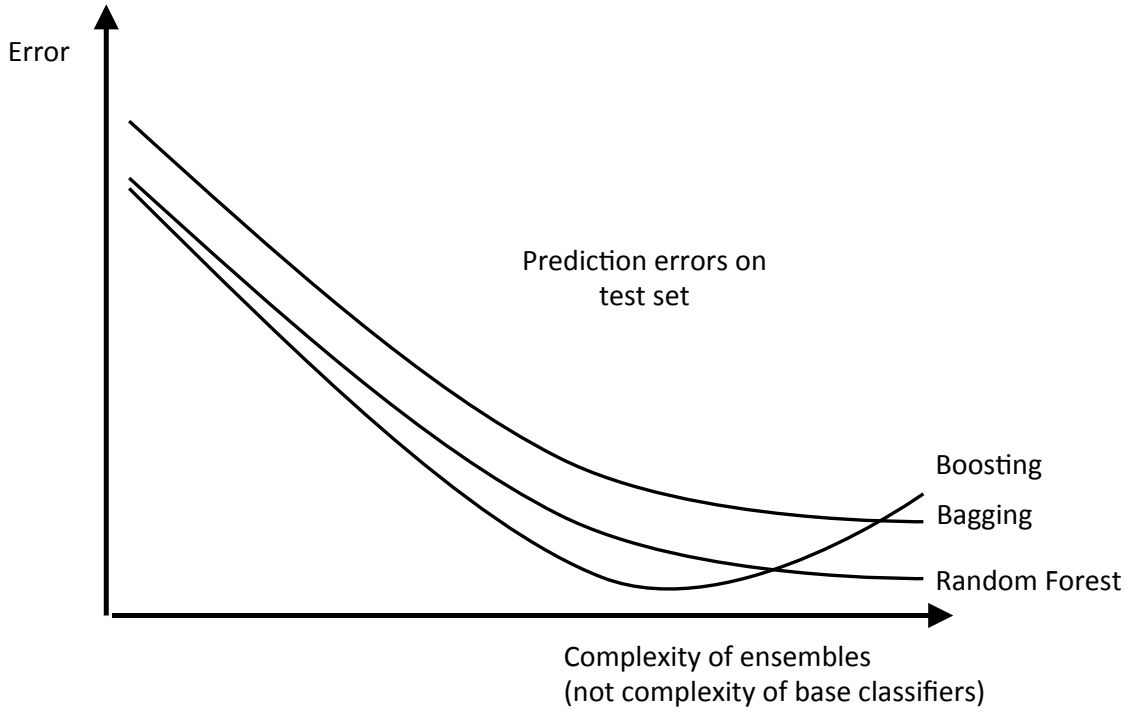


Figure 2.40: The influence of ensemble parameters

2.6.4 Partial dependence plots

The first measures we look at when evaluating our model is AUC and top decile lift. If these measures are extremely high or low this might indicate problems. However, if these measures are in a range that we might expect (e.g., AUC=[0.6-0.9]) there might still be problems with the model. We should also investigate if the relationships on which our model is based are plausible. If we work with a regression technique we might simply look at the coefficients (it has to be noted that multicollinearity will probably make coefficients uninterpretable). But what can we use in case of ensemble methods such as random forest?

While ensemble methods are often credited for creating powerful predictive models, they are discredited for having low interpretability (see Figure 2.41; adapted from Gareth, Witten, Hastie, and Tibshirani 2013) (De Bock and Van den Poel, 2012; Gareth et al., 2013). Neslin et al. (2006) clearly denote that different algorithms are often required depending on whether the goal is description or prediction. Gareth et al. (2013) proclaim that as flexibility of a method increases, its interpretability decreases.

In Gareth et al. (2013) flexibility is synonymous to non-linearity which is a proxy for predictive performance. It is important to note that it might not always pay off to select the most flexible method because more flexibility can translate into a higher potential for overfitting (Gareth et al., 2013, p26).

Model interpretability is advocated by several authors to be one of the main requirements for

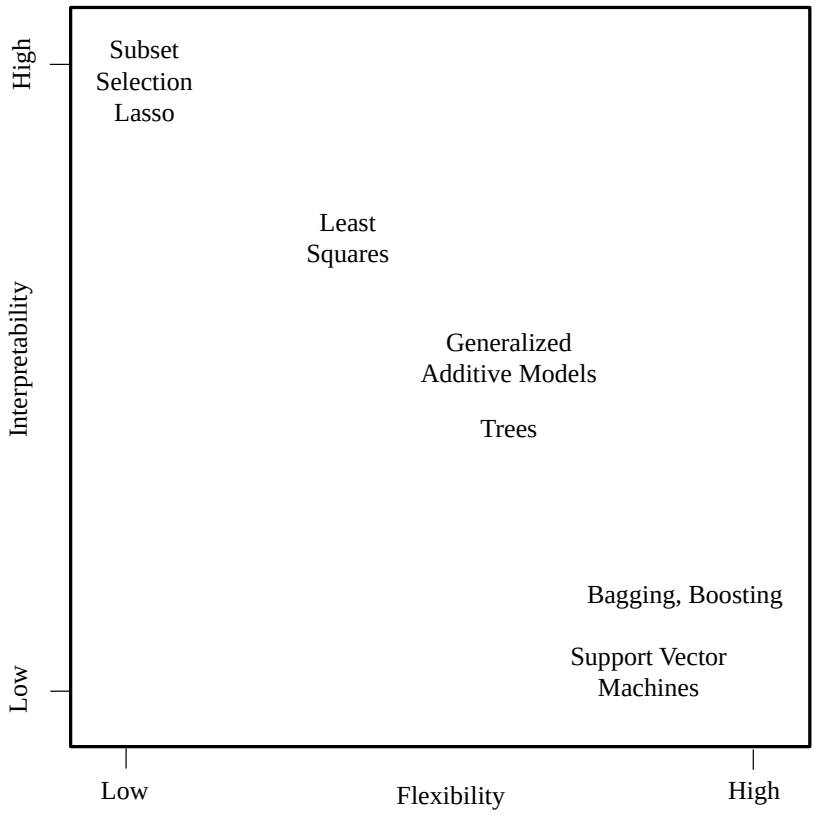


Figure 2.41: Trade-off between flexibility and interpretability

a successful model (Qi et al., 2009; De Bock and Van den Poel, 2012). In Customer Relationship Management, interpretable and intuitive models allow decision makers to gather valuable insights into the drivers of customer behavior (Masand et al., 1999). While it is common knowledge that the importance of predictors can easily be determined using permutation based measures (Breiman, 2001), the criticisms surrounding ensembles about interpretation mainly focus on their incapability of describing the sign or form of the relationship between a predictor and response.

There is however, a method for describing the relationship between a predictive feature and the response using an ensemble model. This method is based on visualization: one of the most informative interpretational tools (Friedman and Meulman, 2003, p24). Partial Dependence Plots (Friedman, 2001; Hastie et al., 2009, Section 10.13.2) can describe the effects of features on the predictions of any black box learning method (Hastie et al., 2009; Cutler et al., 2007).

In general, a classification function h depends on many predictors $x = (x_1, \ldots, x_n)$. Consider a subset x_s containing $p < n$ of those predictors that are of interest to the researcher. The predictors x_c are the complement such that $x = x_s \cup x_c$ and $h(x) = h(x_s, x_c)$. The partial or average dependence of $h(x)$ on x_s is then defined by (Hastie et al., 2009)

$$f_s(x_s) = \frac{1}{N}\sum_{i=1}^{N} h(x_s, x_{ic}) \tag{2.42}$$

where $\{x_{1C}, x_{2C}, \ldots, x_{NC}\}$ are the values of x_c that occur in the training data. This method requires a pass over the data for each set of joint values of x_s (Hastie et al., 2009).

In classification tasks, $h(x)$ is computed on a logit scale. According to Liaw (2014) the underlying reason is that that is where the predictions are closer to symmetric (normal) and homoscedastic, where computing the mean makes sense. Let $P_k(x)$ be the probability of membership of the k^{th} class and N the number of instances. For K-classification the k^{th} response function is then given by Equation 2.43 (Hastie et al., 2009). In case of binary classification Equation 2.43 can be reduced to Equation 2.44.

$$f_s(x_s) = \frac{1}{N}\sum_{i=1}^{N}\left(log\big(P_k(x_s, x_{ic})\big) - \frac{1}{K}\sum_{k=1}^{K} log\big(P_k(x_s, x_{ic})\big)\right) \tag{2.43}$$

$$\begin{aligned} f_s(x_s) &= \frac{1}{N}\sum_{i=1}^{N}\left(log\big(P(x_s, x_{ic})\big) - \frac{1}{2}\Big(log\big(P(x_s, x_{ic})\big) + log\big(1 - P(x_s, x_{ic})\big)\Big)\right) \\ &= \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\left(log\big(P(x_s, x_{ic})\big) - log\big(1 - P(x_s, x_{ic})\big)\right) \\ &= \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\left(logit\big(P(x_s, x_{ic})\big)\right) \end{aligned} \tag{2.44}$$

Thus, the scale in partial dependence plots is half of the logit (log of the odds) of the probability of the event. Please note that Partial Dependence Plot do not follow the convention to take one class as reference category (Berk, 2008, p224). While the overall fit is independent of the choice of reference category, coefficients are affected of by choosing one reference category over the other. Since there is no statistical justification for choosing a particular reference category, and the choice is mostly made on subject matter grounds, the choice varies from researcher to researcher. To avoid these complications, Partial Dependence Plots take the simple mean of the predictions of the K categories as reference. The response variable units are then deviations from the mean or the disparity between the logged predictions for category k and the mean of the logged predictions for all K categories. The units are logits but with the mean over the K categories as reference. Each category has its own equation and its own Partial Dependence Plot. This convention is used even when $K = 2$ and the conventional logit may not result in interpretation issues (Berk, 2008, p224).

From a more practical perspective, Partial Dependence Plots (for ensemble models) are created by the following procedure (Berk, 2008, p222):

1. Grow an ensemble model

2. For a specific predictor x_p, create a Partial Dependence Plot as follows:

   (a) Let v be the number of distinct values of a predictor x_p

   (b) For each value of the v values

      i. Create a novel data set where x_p only takes on that value and leave all other variables untouched

      ii. Predict the response for all instances in that novel data set using the ensemble

      iii. Calculate the mean of the half logit of the predictions yielding one single value for all instances

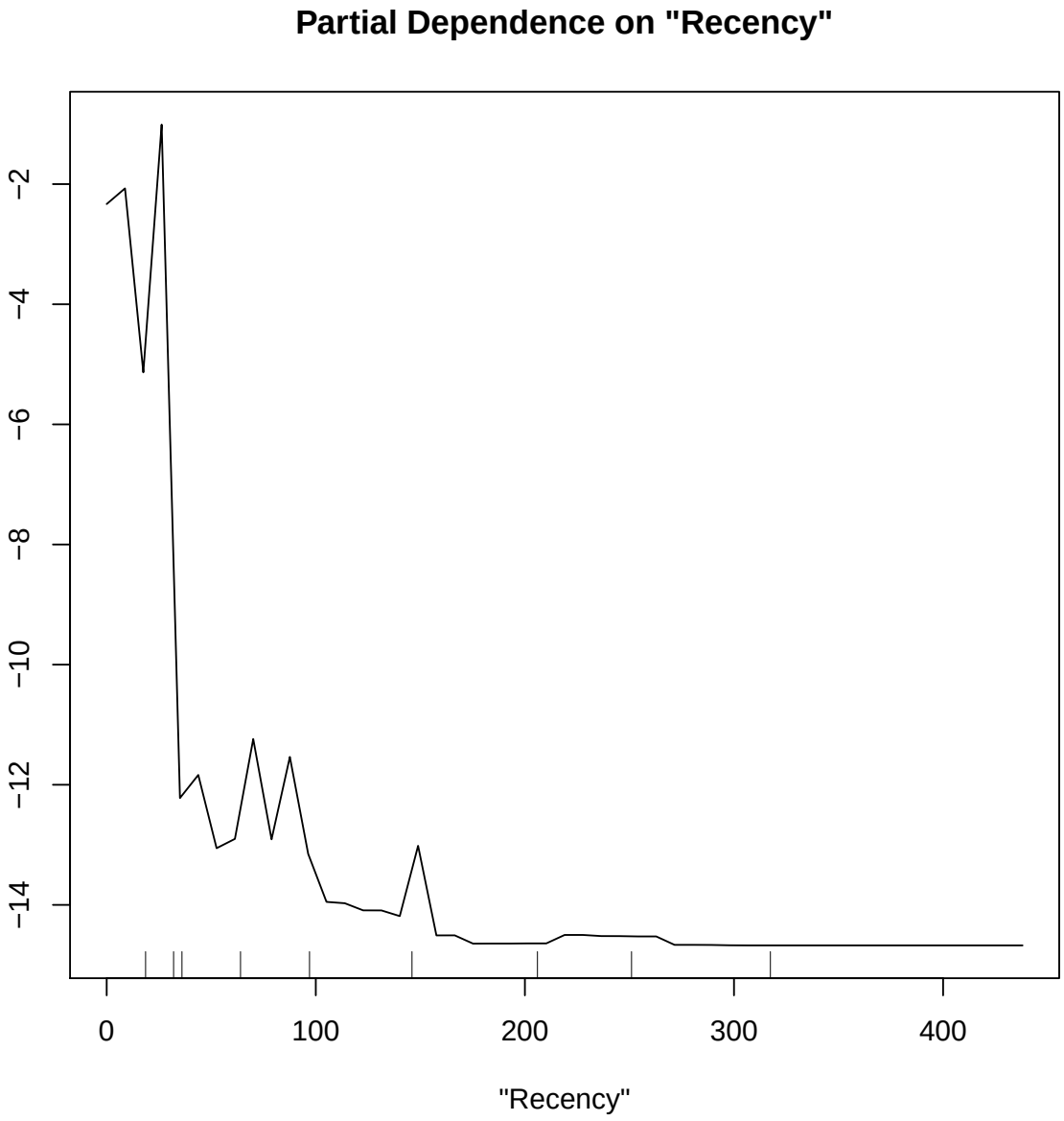
   (c) Plot the mean of half the logit of the predictions against the v values

3. Return to step 2 and repeat for other predictors of interest

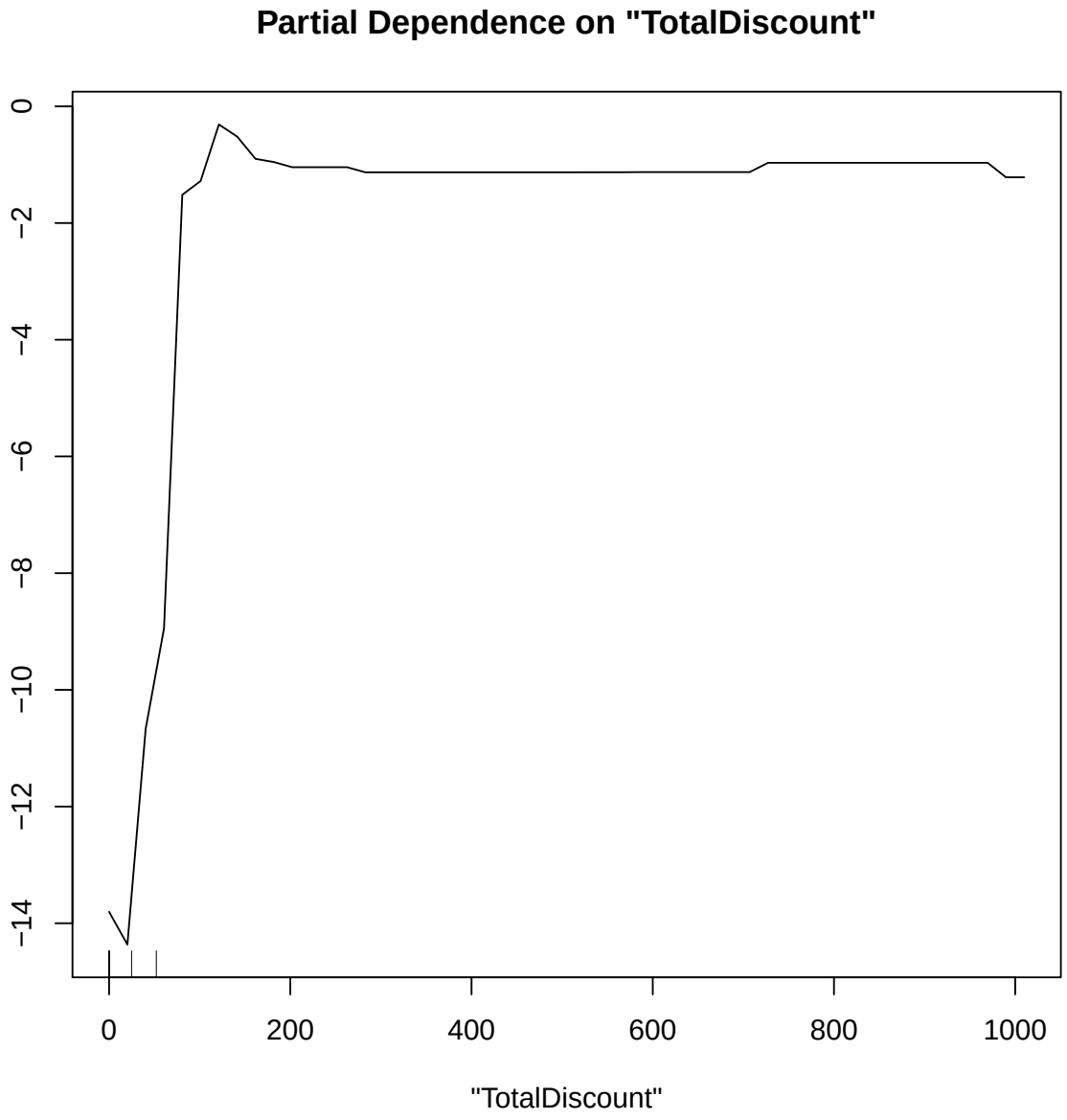
It is important to note that partial dependence plots represent the effect of a predictor or a subset of predictors after accounting for the effects of the other predictors. Analogously to a multiple linear regression, the coefficient of predictor x_1, obtained from regressing y on all x_j, measures the effect of x_1 accounting for the effects of the other variables. In simple linear regression where each x_j is regressed separately on y the coefficient of x_1 ignores the other predictors (Friedman and Meulman, 2003).

Partial dependence plots can be generated using the function `partialPlot` in the `randomForest` package. The following code block contains and example using the model that we built in Section 2.5.11.

```
if (!require('randomForest')){
  install.packages('randomForest',
                   repos='http://crank.rstudio.com',
                   quiet=TRUE)
  require('randomForest')
}

#-# Loading required package: randomForest
#-# randomForest 4.6-12
#-# Type rfNews() to see new features/changes/bug fixes.

# The two top predictors were Recency and TotalDiscount.
# Partial dependence plots for these two variables can
# be created as follows:

#The which.class=1 parameter is very important. The default is to take
#the reference category to be 0 and is never what we want.

partialPlot(x=rFmodel,x.var="Recency",
            pred.data=BasetableTEST,which.class=1)
```




```
#From the partial dependence plot we see a small recency
#results in a higher propensity to have a 1.

partialPlot(x=rFmodel,x.var="TotalDiscount",
            pred.data=BasetableTEST,which.class=1)
```




```
#In contrast a higher value for TotalDiscount results in
#higher propensity to have a 1.

#The advantage of partial dependence plots is that
#they display the nonlinearity of the relationship.
```

2.7 Model Deployment

Model deployment consists in anything related to making the code run in the deployment environment. Some companies like to run the code on their on own servers and maintain the code themselves. Other companies like a full service model in which they have another company store and analyze their data. An intermediate solution is when a CRM company has access to a company's data, or gets data dumps, and runs the code on the CRM company's servers. In the latter case the CRM company maintains the code, fits models, deploys the models, and provides the results (e.g., a customer ID and a propensity score) to the company. Whichever option the company chooses mainly depends on whether the company has or wants to invest in computing machinery and whether they have the necessary skills to run and maintain the code. If the company does not have the required servers or skills it might make sense to pay an external CRM company for that service, to avoid the high initial cost of equipment and salary costs of specialized employees. In any case, we want our code to be easily maintainable. This Section provides some key concepts to create clean and maintainable code.

2.7.1 Code structure

In all the projects in this book, we need two main functions: a function to build a model, and a function to deploy the model. The function that builds the model will have to do the data preparation, modeling, and model evaluation. The function that deploys the model will have to do data preparation, and make predictions. Most of the code will be contained in the data preparation part and it will be almost identical in both model building and model deployment. The easy way would be to write the data preparation code for the model building function and copy it to the model deployment function. However, copying code is never a good idea. Suppose that we need to change something in the code in the future. This would effectively mean that we need to change it in two places. This process is very error prone. A better way would be to put all the data preparation code in another function and call that function in both the model building function and the model deployment function. This new function can be either created in the model building function and passed along to the model deployment function or created outside the two functions and simply called in those functions.

In the following pseudo code block we provide a possible structure of the model building, model deployment function, and any helper functions that are called in those two functions.

```
# Note: This is pseudo code. Do not run this code.

# The model building function contains four parameters: the start
# of the independent period, the end of the independent period, the
# start of the dependent period and the end of the dependent period.
# We also add an evaluate function that allows the user to decide
# whether the model should be evaluated or not. The verbose parameter
# allows the user to choose whether progress should be printed to
# the screen. Assigning a value to parameters when creating a function
# is equivalent to setting default values. For example, the default
# value of evaluate and verbose is TRUE.
```

```
# modelbuilding <- function(start.ind, end.ind, start.dep, end.dep,
#                   evaluate=TRUE, verbose=TRUE){
#
#   task 1: load packages
#   task 2: call read.and.prepare.data(train=TRUE) function (see below)
#           return list(basetable, result of categories function
#           from dummy package)
#   task 3: if (evaluate==TRUE) split data in training, and test,
#           then train the model, predict, and evaluate
#   task 4: train model on all data
#   task 5: return list(length of independent period, result of
#           categories function from dummy package, model)
#
# }

# The model deployment really needs two parameters, the object
# returned by the modelbuilding function, and the dump date of the
# data. The dump date is the date when the data is exported from
# the database. The dump date is equivalent to the end of the
# independent period in the deployment phase.

# modeldeployment <- function(object, dump.date){
#
#    task 1: load packages
#    task 2: call dataprep <- read.and.prepare.data(
#                                    train=FALSE,
#                                    end.ind=dump.date,
#                                    object$length independent period,
#                                    object$output categories function)
#            return a basetable
#    task 3: call predictions <- predict(object$model,dataprep$basetable)
#    task 4: return dataprep$basetable$customerID, predictions
#
# }

# Note that both functions call the function read.and.prepare.data.
# In the model building function train=TRUE and in the model deployment
# function train=FALSE. This is used to turn on or off certain parts
# in the read.and.prepare.data function. For example in model building
# we want to call the categories function from the dummy package, whereas
# in model deployment we do not call the categories function but we would
# use the output of the categories function applied in model building.
# A similar technique can be applied with reference to computing the
# dependent variable. In model building we would compute the dependent,
# whereas in model deployment we would not.

# read.and.prepare.data <- function(train,...){
#
#    task 1: read in tables
#    task 2: if (train==TRUE) call categories() on all tables
#    task 3: call dummy
```

```
#    task 4: aggregate all tables
#    task 5: merge results of aggregation
#    task 6: impute missing values
#    task 7: return list(basetable, result of categories function
#             from dummy package)
# }
```

2.7.2 Method dispatching

Up to now we have called the two main functions `modelBuilding` and `modelDeployment`. While this is perfectly fine naming functions like this is not according to R conventions. First of all, what if one has code for both customer acquisition and customer churn. If functions have identical names in both applications there is no way to know which function is loaded without looking at the body of the code. Therefore it makes more sense to provide a more specific name. For example, in the acquisition case we might call the model building function `acquisitionModel` and in the churn case we might use the name `churnModel`'. Second, there are other conventions that make a programmer's life easy. For example, to create a plot, one would use the `plot` function, to print to the screen, one would use `print`, to create a summary, one would use `summary`, and to create predictions, one would use `predict`. If R users are used to use `predict` whenever they deploy a model, it does not make sense to call our deployment function `modelDeployment`. We should call it `predict`. However, we cannot just name our model deployment function `predict` because then `predict` would not work anymore for other packages. It would only work for our package and that may create problems when users have multiple packages loaded. The solution lies in what is called method dispatching.

R possesses a generic function mechanism that can be used for an object-oriented programming style. The functions `plot`, `print`, `summary` and `predict` are all generic functions. Each generic function has associated with it a set of other functions, called methods, that have the same arguments as the generic function. Method dispatch takes place based on the class of the object supplied as an argument. If the class does not have a name, or R cannot find the associated method, it will call the default method. For example, consider the following code block.

```
#define two variables
integer_vector <- c(1,1,1,2,2,2)
factor_vector <- as.factor(c(1,1,1,2,2,2))

# Call the generic function called summary.
# summary behaves differently depending on
# whether it is called on a integer or a factor.
# R dispatches (uses) a different method (function)
# based on the class of the object the generic
# function is called on.
summary(integer_vector)

#-#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#-#     1.0     1.0     1.5     1.5     2.0     2.0
```

```r
summary(factor_vector)

#-# 1 2
#-# 3 3

# In the case of the integer vector, R will not
# find a method called summary.integer,
# hence it calls summary.default. In the
# case of the factor vector, R does find a method
# called summary.factor and hence will use that method.
# We can skip the dispatching step and call the methods
# directly.

summary.default(integer_vector)

#-#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#-#     1.0     1.0     1.5     1.5     2.0     2.0

summary.factor(factor_vector)

#-# 1 2
#-# 3 3

# We can even force R to use a method that was not
# designed for the class. For example:

summary.factor(integer_vector)

#-# 1 2
#-# 3 3

# One can print all the methods associated with
# a generic function as follows:
apropos("summary\\.")

#-#  [1] ".__C__summary.table"     "print.summary.table"
#-#  [3] "summary.aov"             "summary.connection"
#-#  [5] "summary.data.frame"      "Summary.data.frame"
#-#  [7] "summary.Date"            "Summary.Date"
#-#  [9] "summary.default"         "Summary.difftime"
#-# [11] "summary.factor"          "Summary.factor"
#-# [13] "summary.glm"             "summary.lm"
#-# [15] "summary.manova"          "summary.matrix"
#-# [17] "Summary.numeric_version" "Summary.ordered"
#-# [19] "summary.POSIXct"         "Summary.POSIXct"
#-# [21] "summary.POSIXlt"         "Summary.POSIXlt"
#-# [23] "summary.proc_time"       "summary.srcfile"
#-# [25] "summary.srcref"          "summary.stepfun"
#-# [27] "summary.table"
```

```
# Because the first argument is a regular expession
# we need to escape (\) the dot which is a metacharacter.
# In R we always need to double backslashes.

# Now let's get back to how we are going to use
# method dispatching. Consider the following
# function called acquisitionModel.
# Imagine that it creates and outputs a model.
# The returned object has class 'acquisition'.

acquisitionModel <- function(x){
 a <- "a model"
 class(a) <- "acquisition"
 a
}

(model <- acquisitionModel())

#-# [1] "a model"
#-# attr(,"class")
#-# [1] "acquisition"

# We now want to create a function called predict,
# because that is what the user is expecting.
# We can find out if there exist already functions
# that are called predict by using 'apropos'.

apropos("predict\\.")

#-# [1] "predict.glm" "predict.lm"

# Indeed there are already several functions.
# Therefore we will need to use method dispatching
# and create a new method by appending the class
# name to the generic function name (i.e., predict).
# Since we have assigned the class 'acquisition' to
# our object, we call our function 'predict.acquisition'.

predict.acquisition <- function(object){
  a <- "predictions from our model"
  a
}

predict(model)

#-# [1] "predictions from our model"

# As we can see, using predict instead of predict.acquisition
# correctly calls predict.acquisition.

# This is equivalent to:
predict.acquisition(model)

#-# [1] "predictions from our model"
```

2.7.3 Making functions invisible

We don't want our helper functions to be visible to the user of our code. We can make them invisible by adding a dot in front of the function name. Consider the following example:

```
rm(list=ls())
ls()

#-# character(0)

# The enivronment is empty.

# We append a dot in front of funcname:
.funcname <- function(x) print(x)
ls()

#-# character(0)

# The environment is still empty, but we
# can call the function if we want:
.funcname(1)

#-# [1] 1

# Without the dot in front of the name:
funcname <- function(x) print(x)
ls()

#-# [1] "funcname"

# Now we can see the function.
```

2.7.4 The '...' argument

The '...' argument is a special argument that allows for flexibility in writing functions. It is said to be flexible because it allows us to refrain from prespecifying a name for additional argument. For example we might have a function that computes the sum of two or more vectors. There will be at least two variables but up front there is no way to know how many additional vectors will need to be summed. So instead of creating parameter names for an unknown number of vectors we can use '...'. The following code block shows that we can have a lot of additional parameters (in this case 10) without specifying their names up front.

```
func <- function(x,y,...){

    sum(x) + sum(y) + sum(...)

}
```

```
func(x=c(1,2,6),y=c(1,2),c(1,2),
     c(1,2,4),c(1,2),c(1,2),c(1,2),
      c(1,2),c(1,2),c(1,2,9),c(1,2),
      c(1,20))

#-# [1] 73
```

Now, in helper functions this might not make immediate sense, since we will be the only users of the those functions and we can adapt the function (and its parameter values) whenever we want. However, there is one advantage in using '...' instead of formal parameter names: it can help in understanding the code. Helper functions get called from different main functions, and different parameters (not parameter values) might be required depending on which main function calls the helper function. For example, when calling `read.and.prepare.data` from the `predict` function there will be a parameter with dummy variables categories. When calling `read.and.prepare.data` from the model building function we don't need this parameter because categories are learned on the data in that function. Therefore it might make sense to incorporate that parameter in the '...' argument as it might confuse whoever is maintaining the code afterwards that has not been involved in code development. Using '...' signals that a given parameter is not always required.

For example, whenever a new programmer looks at our code, he or she will first look at the main functions. Because the model building phase comes first, the `modelBuilding` function will be inspected first. While walking through that function the programmer will encounter a call to a helper function (e.g., `read.and.prepare.data`). This will prompt the user to look into that function. If there are fifteen parameters the programmer will first investigate all these parameters, even if only the first one is required to run the code. Those fourteen additional parameters might only make sense from the perspective of the `predict` function.

In sum, it is good practice to investigate all the named parameters of a function before using it. However some parameters might only be understood properly when looked at from a given context. For example the parameters of a helper function might only be understood when looked at from the perspective of the function that calls the helper function. Therefore it might make sense to put certain parameters in the special parameter denoted by '...'.

```
# Consider the first main function: mainFunc1.
# It calls the helper function and creates a value.
# Suppos that runif(1) is the result from some complex
# coding.

# x: scalar
# y: scalar
mainFunc1 <- function(x,y){
  helped <- helper(x,y)
  list(helped,runif(1))
}

# What is helper doing with those x and y?
```

```r
# Looking at the function we see that it is a sum.
# There might be additional parameters but since they
# are in ... we do not have to worry about them.
# The ... indicate that they are only required in
# specific contexts and not important in all context.
# Since mainFunc1 only uses the two first named
# parameters we do not need to worry about '...'.

# x: scalar
# y: scalar
helper <- function(x,y,...){
    x + y + (if (length(list(...)$z)==0) 0 else list(...)$z)
}
# Note that the expression list(...) returns a named list
# of the components of ...

# We then look at the second main function mainFunc2.
# We see that there is an additional parameter called
# z in helper. This parameter is seems to be relevant
# only after mainFunc1 is called, and hence only relevant
# when calling mainFunc2.

# x: scalar
# y: scalar
# object: output mainFunc1
mainFunc2 <- function(x,y,object){
  helper(x,y,z=object[[2]])
}

(outmain1 <- mainFunc1(x=1,y=2))

#-# [[1]]
#-# [1] 3
#-#
#-# [[2]]
#-# [1] 0.8097823

(outmain2 <- mainFunc2(x=1,y=2,object=outmain1))

#-# [1] 3.809782

# Now, it is important to understand that
# the helper function above is equivalent to the following
# function.
helper <- function(x,y,z=0){
    x + y + z
}

(outmain2 <- mainFunc2(x=1,y=2,object=outmain1))

#-# [1] 3.809782
```

```
# The only difference is that the person that tries to
# understand the code will not be prompted
# to try and figure out what the additional parameter is
# all about (when calling helper from mainFunc1) until it
# is actually used later in the code (when calling helper from
# mainFunc2). This can speed up any efforts to try and
# understand the code. This small example might not be very convincing,
# but when there are a lot of additional parameters it is key
# to see them from the perspective of the calling function
# to be able to understand them efficiently and properly.
```

2.7.5 Environments

The code in this section is available from:
`http://ballings.co/hidden/aCRM/code/chapter2/environments.R`

The absolute goal is to create code that is easy to maintain. Consider the situation where we have (almost) identical code in two functions. If we need to change something in that code in the future, we would have to make that change twice. This does not qualify as easily maintainable code. The solution is to put the identical code in another function, and call that function in the two functions. This means that if the code requires a change, we would only have to make that change once. A problem might occur, however, when objects are inherited from the enclosing environment. Consider the following `helper` and `mainFunc` functions. The function `mainFunc` calls the helper function. What do you expect a call to `mainFunc()` to return?

```
helper <- function(){
  x
}
x <- 1

mainFunc <- function() {
 x <- 2
 helper()
}

mainFunc()

#-# [1] 1
```

In contrast to what we might expect `mainFunc()` returns the value 1, and not 2. This brings us to the definition of a function. A function has three basic components: a formal argument list (`formals()`), a body (`body()`), and an environment (`environment()`) (R Core Team, 2015c). A function's environment (R Core Team, 2015b) is the environment that was active when the function was defined (i.e., created). The environment contains a pointer to an enclosing environment, also called enclosure, or parent environment. Symbols (variables) in the enclosure are attached to the function's body and argument list. Thus, when a function is called, and a variable is requested in the function, it is first sought locally inside the function, then in the

enclosure (i.e., the enclosing environment when the function was defined), the enclosure of the enclosure, etc (R Core Team, 2015d). For example, in the code block above, the `helper` function is defined in the global environment and hence contains a pointer to the global environment and its enclosure. When `helper` is subsequently called in `mainFunc` the variable `x` is requested in `helper`. The function `helper` will first look inside itself to see if `x` is defined there. It is not, so `helper` will look in its enclosing environment (the environment when it was defined), and will find `x` to be 1. We can easily find out what the chain of environments is that the function will walk through using the following code:

```
envir <- environment(helper)
repeat {
  if (environmentName(envir)!="") {
    cat(environmentName(envir),"\n")
  } else {
    cat(str(envir, give.attr=F))
  }
  if (environmentName(envir) == 'R_EmptyEnv') break
  envir <- parent.env(envir)
}

#-# R_GlobalEnv
#-# package:randomForest
#-# package:lift
#-# package:rotationForest
#-# package:AUC
#-# package:data.table
#-# package:knitr
#-# package:stats
#-# package:graphics
#-# package:grDevices
#-# package:utils
#-# package:datasets
#-# package:methods
#-# Autoloads
#-# base
#-# R_EmptyEnv
```

We see that `helper` first looks in `R_GlobalEnv`, then goes and look in `tools:rstudio`, ..., all the way up to `R_EmptyEnv`. `R_EmptyEnv` is always the last environment to be searched and as the name indicates, is empty, effectively ending the search. If the first environment is `R_GlobalEnv` the above code block will be equivalent to `search`.

Consider another example in the following code block. What do you expect to be the output of `output$prepareData()`?

```
BuildModel <- function(){
  x <- 1
  prepareData <- function(){
    x
```

```
  }
  model <- "a model"
  list(model=model,prepareData=prepareData)
}

output <- BuildModel()

output$prepareData()

#-# [1] 1
```

Indeed, `output$prepareData()` returned the value 1. Now, what do you think `DeployModel(object=output)` will return? Will it be 1 or 2?

```
DeployModel <- function(object){
  x <- 2
  output$prepareData()
}

DeployModel(object=output)

#-# [1] 1
```

The value of `DeployModel(object=output)` is not 2, but 1. This is generally not what we want. The solution is to change the environment of `prepareData` as follows:

```
DeployModel <- function(object){
  environment(object$prepareData) <- environment()
  x <- 2
  object$prepareData()
}

DeployModel(object=output)

#-# [1] 2
```

Now the value is what we want it to be: 2. To see what is going on we modified the above code block by printing additional information:

```
DeployModel <- function(object){
  cat("Before changing environment:\n")
  cat("   Environment name: ")
  print(environment(object$prepareData))
  cat("   Objects in environment: ")
  cat(ls(environment(object$prepareData)),"\n")
  cat("   Value of x:",
    get("x",envir=environment(object$prepareData)),
    "\n")
```

```
  environment(object$prepareData) <- environment()
  x <- 2

  cat("After changing environment:\n")
  cat("   Environment name: ")
  print(environment(object$prepareData))
  cat("   Objects in environment: ")
  cat(ls(environment(object$prepareData)),"\n")
  cat("   Value of x:",
    get("x",envir=environment(object$prepareData)),
    "\n")

  cat("Result: \n")
  object$prepareData()
}


DeployModel(object=output)

#-# Before changing environment:
#-#    Environment name: <environment: 0x7fa72bfc6718>
#-#    Objects in environment: model prepareData x
#-#    Value of x: 1
#-# After changing environment:
#-#    Environment name: <environment: 0x7fa728dde878>
#-#    Objects in environment: object x
#-#    Value of x: 2
#-# Result:
#-# [1] 2
```

To solidify your understanding, make the exercise in Section ??.

3

Case study: Customer Acquisition

Deliverables:

- Data dictionary
- ERD
- Code flowchart
- One function containing all the code to create a model
- One function containing all the code to make predictions using the model

The quality of the two functions depends on performance (top decile lift and AUC)(75%), code maintainability (shorter code is better)(15%), and run time (faster is better)(10%).

All code in this chapter can be downloaded from:
`http://ballings.co/hidden/aCRM/code/chapter3/acquisition.R`

3.1 Business Understanding

A start-up tech company has launched its primary product a couple of months ago. The product is a productivity suite with apps and desktop clients for all major platforms (iOS, Android, Windows Phone, Windows, OSX, Linux). Everything is synced across platforms. The base version has limited functionality and is perfect for individual users. The premium version adds team and admin features for business users. Before one can download the app and use the service a user needs to fill out a form about their work place or employer and provide contact information. The company's strategy is based on the belief that if a sufficient number of team members start using

the product, there will eventually be a tipping point where purchasing those premium features would become very attractive.

The company has been receiving a lot of registrations from companies to use their base version but sales of the premium version are not suffient to sustain the company's current cash burn rate. The company is asking our help with converting users to customers. The company has a small sales team of three sales reps that are using the registration data to call companies at random. The team has not had a lot of success lately. Our job is to build an acquisition model to predict which company is most likely to become a customer. We can then rank the companies by their acquisition propensity score and the sales team can then call the companies most likely to become a customer.

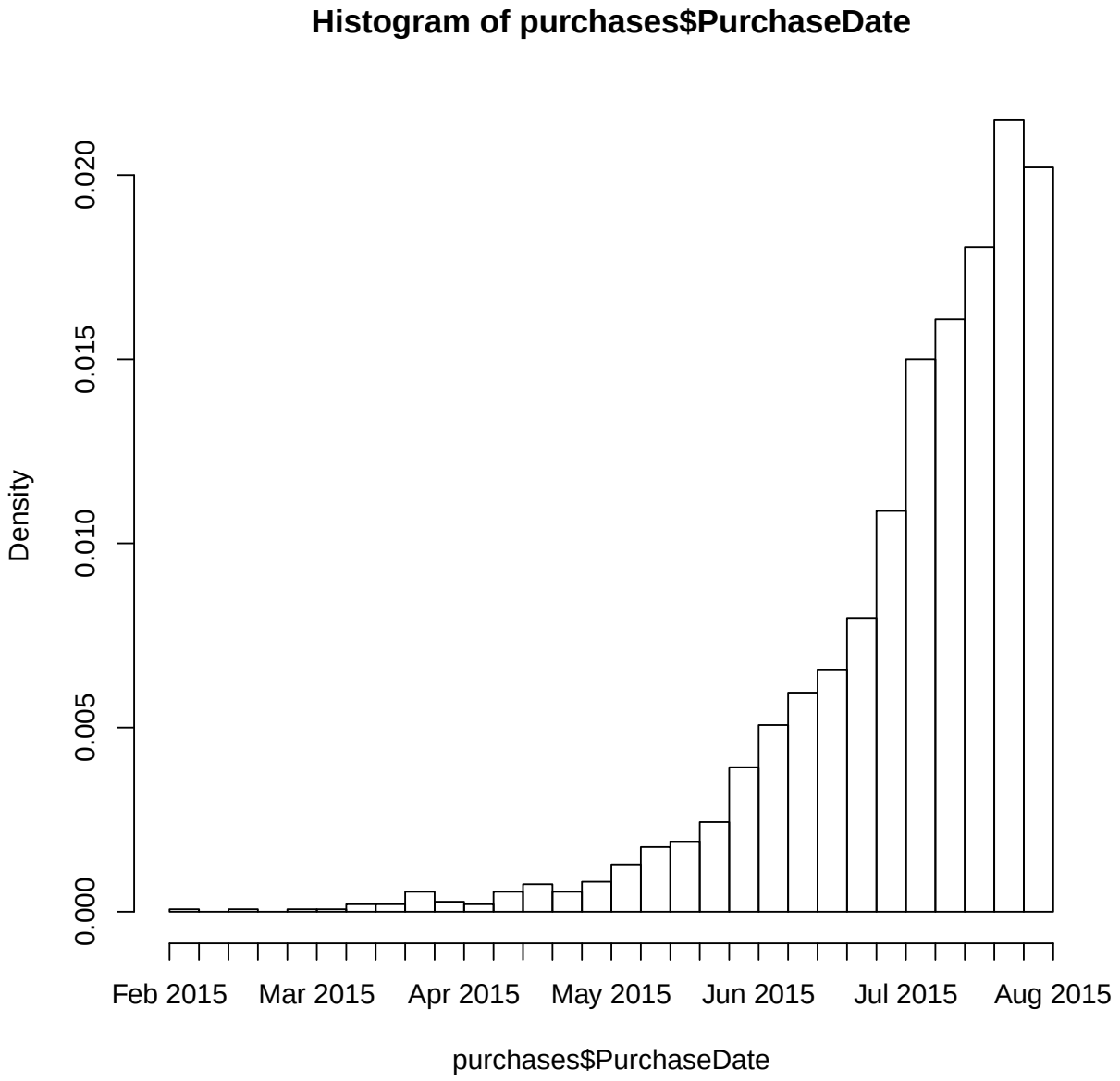
3.2 Data Understanding

```
#Read the data using the read.csv function
customers <-
  read.csv( "http://ballings.co/hidden/aCRM/data/chapter3/customers.csv",
            header=TRUE,
            colClasses="character")

purchases <-
  read.csv("http://ballings.co/hidden/aCRM/data/chapter3/purchases.csv",
           header=TRUE,
           colClasses=c("character",
                        "Date",
                        "character",
                        "numeric"))

registrations <-
  read.csv("http://ballings.co/hidden/aCRM/data/chapter3/registrations.csv",
           header=TRUE,
           colClasses=c("character",
                        "character",
                        "character",
                        "character",
                        "Date"))


#Look at the data
str(customers, vec.len=1)

#-# 'data.frame': 2114 obs. of  5 variables:
#-#  $ CustomerID    : chr  "1" ...
#-#  $ ContactName   : chr  "Mbengue, Randy" ...
#-#  $ CompanyName   : chr  "Action Wars" ...
#-#  $ CompanyAddress: chr  "474 Orchard Avenue, Moorhead, MN 56560" ...
#-#  $ PhoneNumber   : chr  "8907248356" ...

str(purchases, vec.len=1)
```

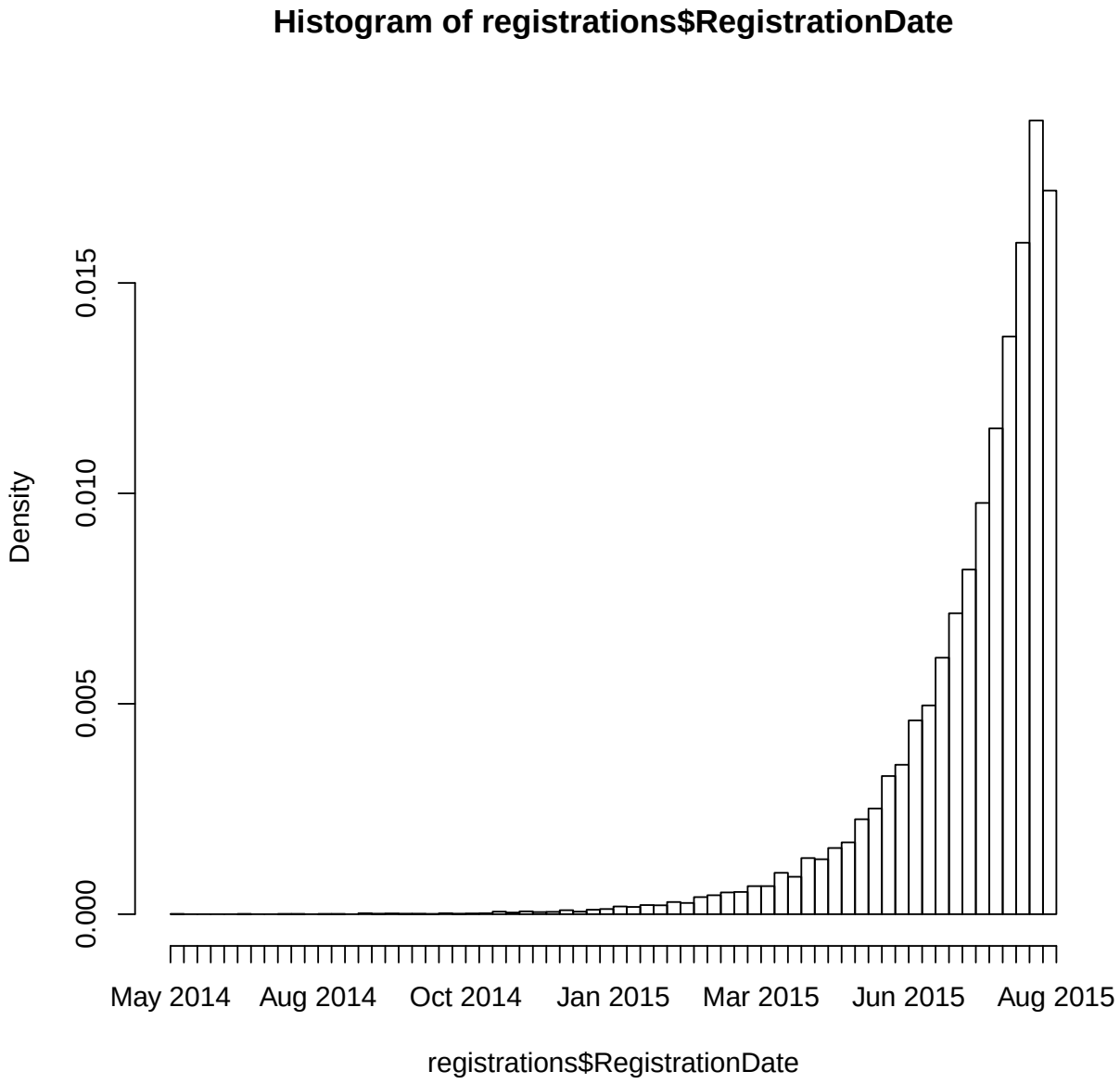
```
#-# 'data.frame': 2114 obs. of  4 variables:
#-#  $ CustomerID   : chr  "1" ...
#-#  $ PurchaseDate : Date, format: "2015-02-05" ...
#-#  $ NumberUsers  : chr  "2" ...
#-#  $ PurchasePrice: num  99.9 ...

str(registrations, vec.len=1)

#-# 'data.frame': 25714 obs. of  5 variables:
#-#  $ ContactName     : chr  "Bonner, Braydon" ...
#-#  $ CompanyName     : chr  "Action Academy" ...
#-#  $ CompanyAddress  : chr  "1112 3rd Drive North, Woburn, MA 01801" ...
#-#  $ PhoneNumber     : chr  "6193847620" ...
#-#  $ RegistrationDate: Date, format: "2014-05-31" ...

#Obtain information required for the ERD

#Relationship 1
#purchases and customers can be linked through CustomerID
#What are the characteristics of this relationship?

#Is CustomerID unique in both tables?
#It should be in customers,
#but we expect it won't in purchases.
length(customers$CustomerID) ==
    length(unique(customers$CustomerID))

#-# [1] TRUE

#Yes it is unique in customers
length(purchases$CustomerID) ==
    length(unique(purchases$CustomerID))

#-# [1] TRUE

#It is also unique in purchases. This means
#each customer has only made one purchase up to now.
#This is plausible since it is a new product

#Does each customer have one purchase?
all(customers$CustomerID %in% purchases$CustomerID)

#-# [1] TRUE

#Yes

#In sum
#purchases 1 CustomerID 1 customers
#1 purchase has 1 customer
#1 customer has 1 purchase
#they are linked by CustomerID
```

```r
#This means that we could simply merge customers and
#purchases without aggregating

#Relationship 2:
#customers and registrations can be linked
#through CompanyName. Is CompanyName unique
#in customers?
length(customers$CompanyName) ==
  length(unique(customers$CompanyName))

#-# [1] TRUE

#Yes

#Is CompanyName unique in registrations?
length(registrations$CompanyName) ==
  length(unique(registrations$CompanyName))

#-# [1] FALSE

#No. This is was we expected since individual
#users of companies are registering

#How many registrations does a customer typically have?
reg_agg <- aggregate(registrations$CompanyName,
                     by=list(CompanyName=registrations$CompanyName),
                     length)
merged <- merge(customers[,"CompanyName",drop=FALSE],
                reg_agg,by="CompanyName",all.x=TRUE)
head(merged)

#-#        CompanyName x
#-# 1      Action Wars 2
#-# 2     Air Missions 1
#-# 3       Air Square 2
#-# 4   Airport Family 2
#-# 5  Airport Hunters 3
#-# 6 Airport Princess 2

summary(merged$x)

#-#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
#-#   1.000   1.000   2.000   2.614   3.000   8.000       5

#We see that there are NAs. This means that there are 0's
#We can safely say that the relationship is 0..N.
#This means there are customers that don't have registrations
#It is unclear how this could happen. Mabye a data quality problem.
#There are only 5 NAs: it might be that some companies received test
# accounts. These are the customers that don't have a registration.
merged[is.na(merged$x),]
```

```
#-#            CompanyName  x
#-# 145      Backyard Boss NA
#-# 366        Box Brother NA
#-# 464      Casino Border NA
#-# 756 Coral Intervention NA
#-# 824       Dance School NA

#How many customers does a registration have? This should
#be 0 or 1 since registrations are done by employees and
#it is the employer who buys the subscription
#Let's check
merged <- merge(data.frame(customers[,"CompanyName",drop=FALSE],x=1),
                registrations[,"CompanyName",drop=FALSE],
                by="CompanyName",
                all.y=TRUE)
head(merged)

#-#        CompanyName  x
#-# 1   Action Academy NA
#-# 2   Action Academy NA
#-# 3 Action Assistant NA
#-# 4 Action Assistant NA
#-# 5 Action Assistant NA
#-# 6 Action Assistant NA

summary(merged$x)

#-#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
#-#       1       1       1       1       1       1   20201

#Indeed, there are NA's so this means that not every
#registration has a customer, and the only other value
#is a 1. Hence it is a 0..1 relationship.

#In sum
# customers 0..1 CompanyName 0..N registrations

#Let's have a lookg at the frequency of purchases
#and registrations over time. Registrations and purchases
# have definitely picked up.
hist(purchases$PurchaseDate,
     breaks="weeks",format = "%b %Y")
```




```
hist(registrations$RegistrationDate,
     breaks="weeks",format = "%b %Y")
```




```
range(purchases$PurchaseDate)

#-# [1] "2015-02-05" "2015-08-30"

range(registrations$RegistrationDate)

#-# [1] "2014-05-31" "2015-08-30"

#How are registration date en purchase date related?
mer <- merge(registrations[,c("RegistrationDate","CompanyName")],
             customers[,c("CompanyName","CustomerID")],
```

```
            by="CompanyName")
mer <- merge(mer,purchases[,c("PurchaseDate","CustomerID")],
            by="CustomerID")
str(mer,vec.len=1)

#-# 'data.frame': 5513 obs. of  4 variables:
#-#  $ CustomerID      : chr  "1" ...
#-#  $ CompanyName     : chr  "Action Wars" ...
#-#  $ RegistrationDate: Date, format: "2015-01-04" ...
#-#  $ PurchaseDate    : Date, format: "2015-02-05" ...

summary(as.integer(mer$PurchaseDate- mer$RegistrationDate))

#-#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#-#   32.00   32.00   32.00   32.01   32.00   33.00

# This is very peculiar, this means that all employees of a company
# always registered on the exact same day. Moreover that day is always
# 32 days before the actual purchase date. It could be that there is/was
# a trial period of 32 days but it doesn't explain why all employees
# always register on the same day. Is there a script running
# in the database that resets the registration date to the minimum
# or maximum registration date within a company? Also, are these
# free trials of the premium version? Or is this the base version?
# If we have answers to these questions we potentially simplify our modeling,
# but because we haven't we cannot do that at this point.
```

Tables 3.1, 3.2, and 3.3 contain the data descriptions, also called data dictionary. Data dictionaries are valuable in two ways: (1) sometimes variable names are unclear and require explanation, and (2) having a physical copy of the table contents is always handy because it alleviates the need to always look back at the data using code. When working on big projects it is recommended to print out the data dictionary and have it on your desk. Figure 3.1 contains the ERD of the acquisition database.

Table 3.1: Data description of the customers table

Variable name	Description
CustomerID	Customer ID
ContactName	The name of the contact who made the actual purchase. Probably the team leader
CompanyName	Name of the company that made the purchase
CompanyAddress	Company address
PhoneNumber	Phone number of the contact name

Table 3.2: Data description of the purchases table

Variable name	Description
CustomerID	Customer ID
PurchaseDate	Data of purchase
NumberUsers	How many users can use the premium software / service
PurchasePrice	How much was paid in total

Table 3.3: Data description of the registrations table

Variable name	Description
ContactName	Contact name
CompanyName	Company name
CompanyAddress	Company address
PhoneNumber	Phone number of the contact name
RegistrationDate	Date of registration

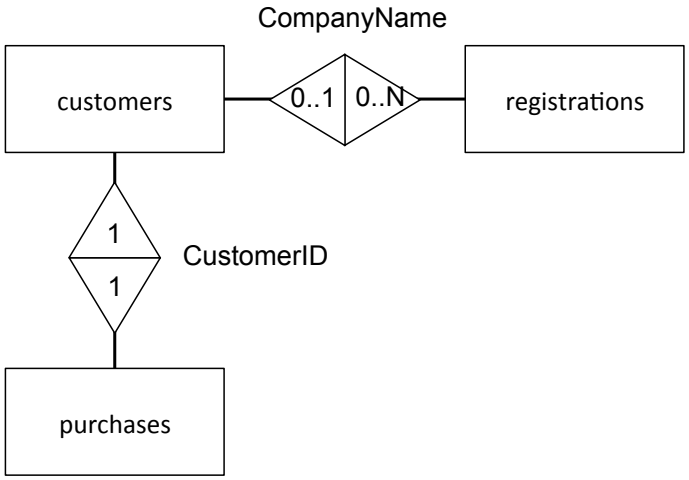


Figure 3.1: ERD of the acquisition database

3.3 Data Preparation

3.3.0.1 Code flowchart

Figure 3.2 contains the flowchart. It is important to first split up the data in data from the independent period and data from the dependent period. Next we compute five predictors and the dependent variable. Finally we merge all predictors and the dependent.

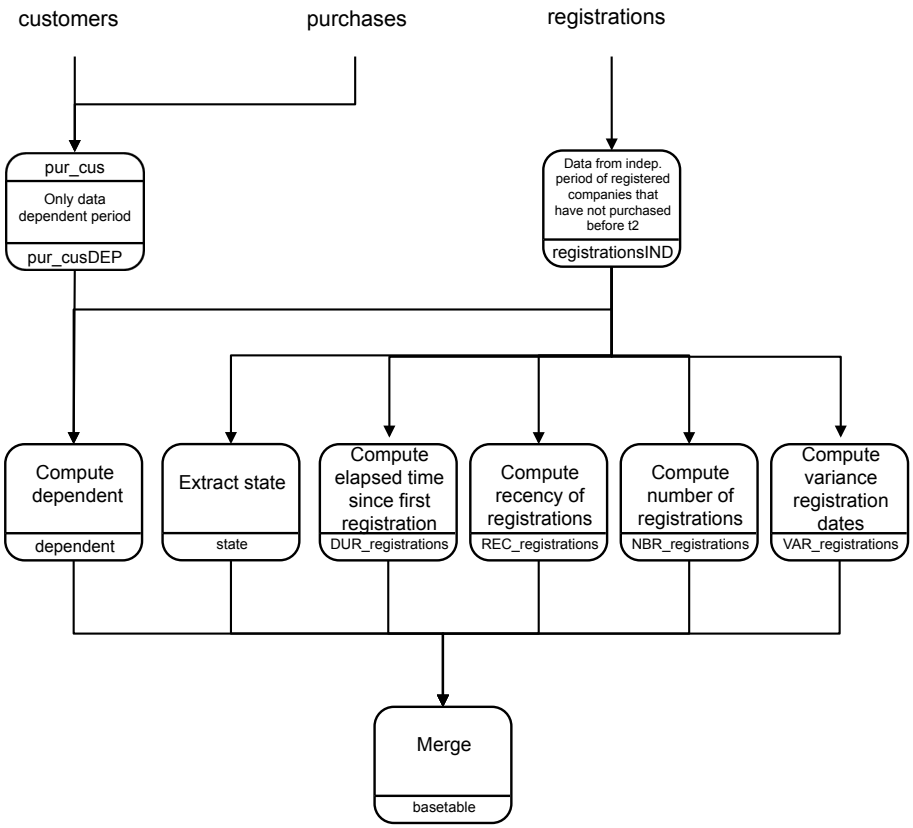


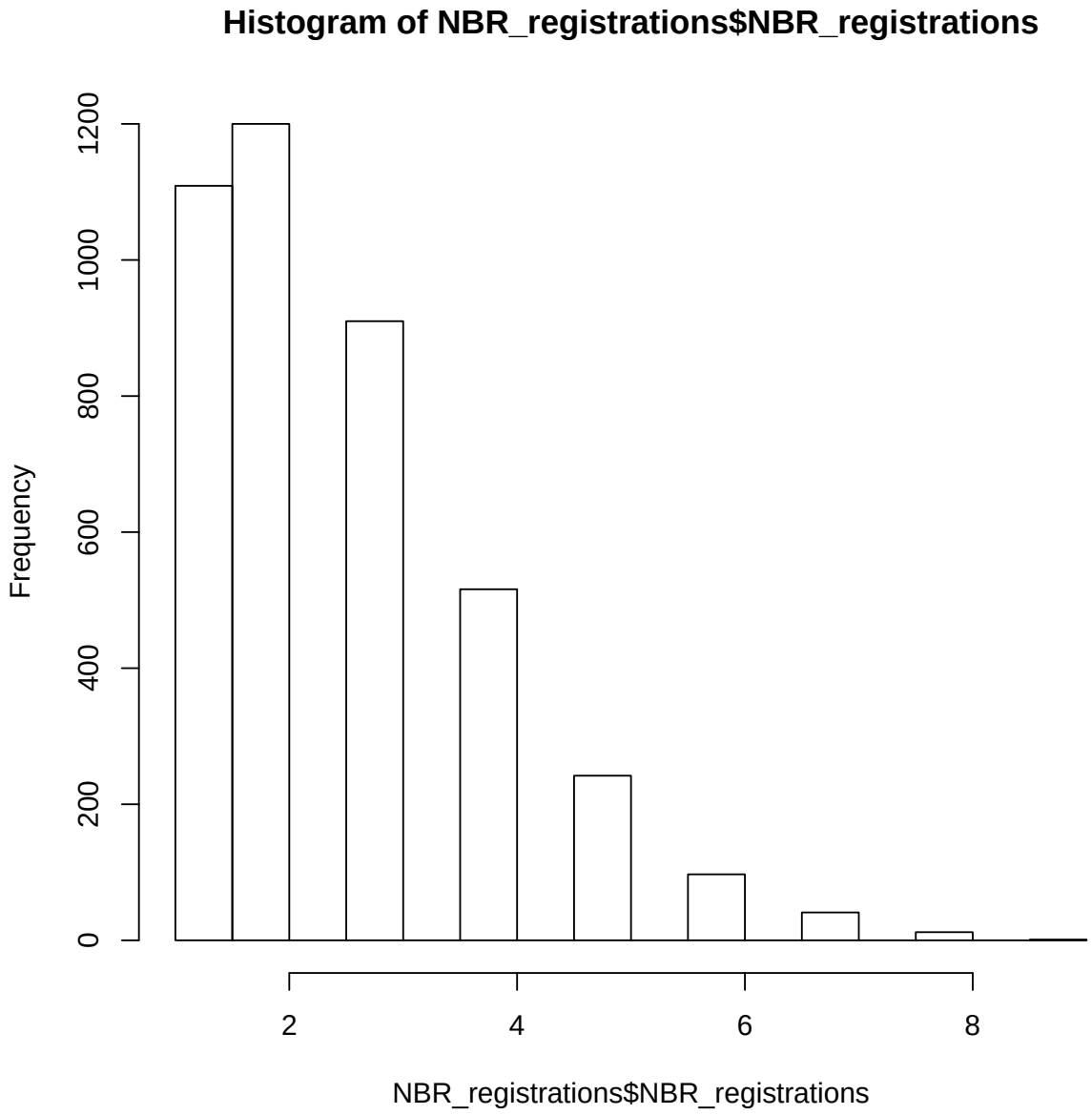
Figure 3.2: Code flowchart for the acquisition project

3.3.0.2 Code

```
#The first question to answer is always: do we need
#all available tables?
#At first sight we need all tables:
#customers: to compute the dependent,
#registrations: to compute the predictors
#purchases: for the time window (purchase date)

# Wich variables will we compute?
#dependent:
```

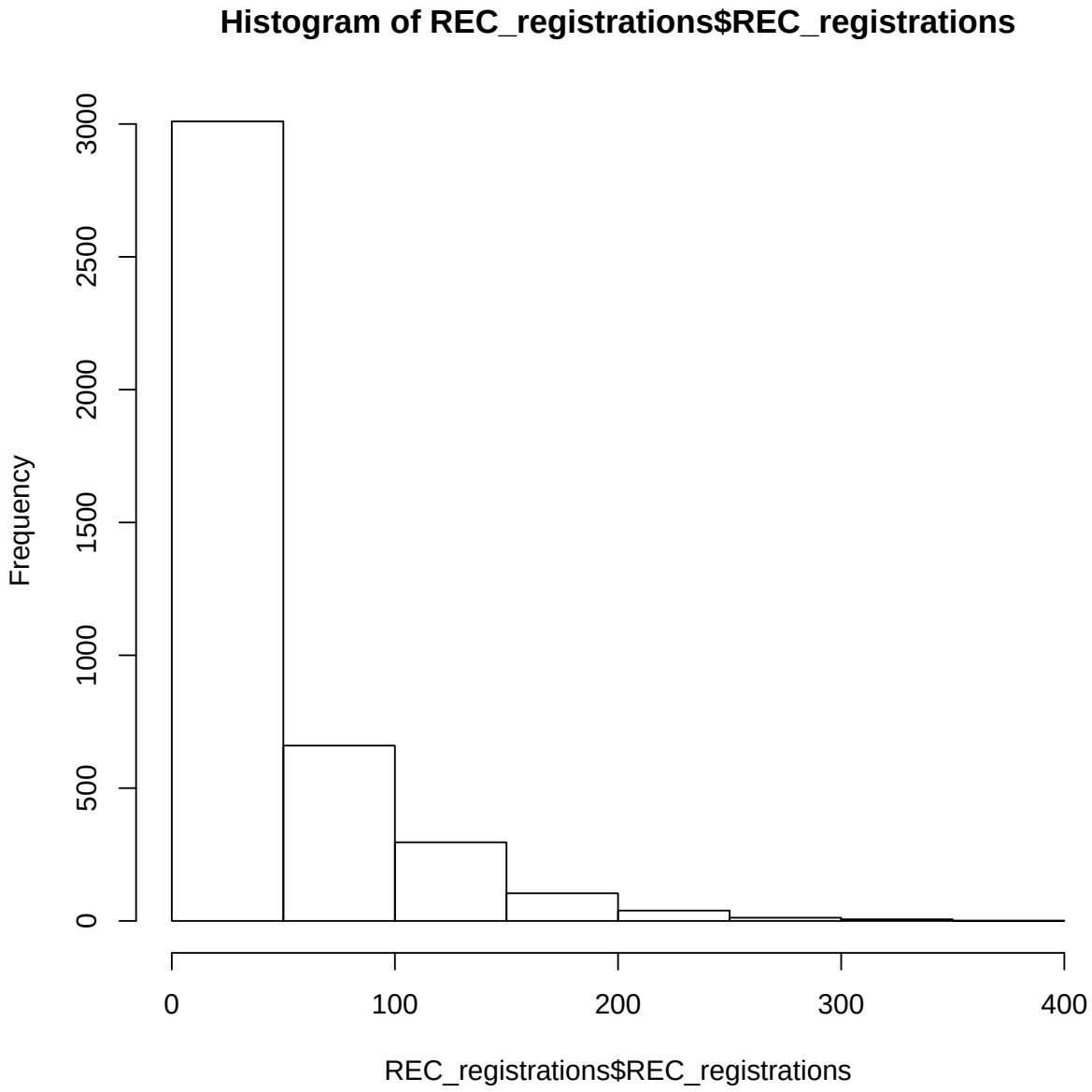
```
#   acquisition: Is company in the customer table. If yes: acquired.
#independents:
#   number of registrations
#   recency of registration
#   elapsed time since first registration
#   variance registration time
#   state
#   zip code first three digits

#Because they have a 1:1 relationship, we can merge
#purchases and customers
pur_cus <- merge(purchases,customers,by="CustomerID")

# Create time window
#Set t4 to the maximum purchase date
(t4 <- max(pur_cus$PurchaseDate))

#-# [1] "2015-08-30"

#The software company wants a dependent
#period of 30 days,
(t3 <- t4-30)

#-# [1] "2015-07-31"

#Use an operational period of 1 day
(t2 <- t3 - 1)

#-# [1] "2015-07-30"

#Set t1 to the minimum registration data
(t1 <- min(registrations$RegistrationDate))

#-# [1] "2014-05-31"

#Split data into independent and dependent data
registrationsIND <- registrations[registrations$RegistrationDate <= t2 &
                    registrations$RegistrationDate >= t1,]

pur_cusDEP <- pur_cus[pur_cus$PurchaseDate > t3 &
                      pur_cus$PurchaseDate <= t4,]

#Make sure to only select companies that have not purchased before t2
pur_cus_bef_t2 <- unique(pur_cus[pur_cus$PurchaseDate <= t2,"CompanyName"])

registrationsIND <-
  registrationsIND[!registrationsIND$CompanyName %in%
                   pur_cus_bef_t2,]

#Compute dependent
#This comes down to merging registrationsIND with pur_cusDEP
```

```
#Note the left outer merge
dependent <-
  merge(data.frame(CompanyName=unique(registrationsIND[,"CompanyName"]),
                   stringsAsFactors=FALSE),
        data.frame(pur_cusDEP[,"CompanyName",drop=FALSE],Acquisition=1),
        by="CompanyName", all.x=TRUE)

dependent$Acquisition[is.na(dependent$Acquisition)] <- 0

dependent$Acquisition <- as.factor(dependent$Acquisition)

str(dependent, vec.len=1)

#-# 'data.frame': 4128 obs. of  2 variables:
#-#  $ CompanyName: chr  "Action Academy" ...
#-#  $ Acquisition: Factor w/ 2 levels "0","1": 1 1 ...

table(dependent$Acquisition, useNA="ifany")

#-#
#-#    0    1
#-# 2915 1213

#Compute predictor variables
#We can only use the registrationsIND table
str(registrationsIND, vec.len=1)

#-# 'data.frame': 10487 obs. of  5 variables:
#-#  $ ContactName     : chr  "Bonner, Braydon" ...
#-#  $ CompanyName     : chr  "Action Academy" ...
#-#  $ CompanyAddress  : chr  "1112 3rd Drive North, Woburn, MA 01801" ...
#-#  $ PhoneNumber     : chr  "6193847620" ...
#-#  $ RegistrationDate: Date, format: "2014-05-31" ...

#   Number of registrations

NBR_registrations <- aggregate(registrationsIND[,"CompanyName",drop=FALSE],
                               by=list(CompanyName=registrationsIND$CompanyName),
                               length)

colnames(NBR_registrations)[2] <- "NBR_registrations"

str(NBR_registrations, vec.len=1)

#-# 'data.frame': 4128 obs. of  2 variables:
#-#  $ CompanyName      : chr  "Action Academy" ...
#-#  $ NBR_registrations: int  2 5 ...

summary(NBR_registrations$NBR_registration)

#-#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#-#    1.00    1.00    2.00    2.54    3.00    9.00

hist(NBR_registrations$NBR_registrations)
```



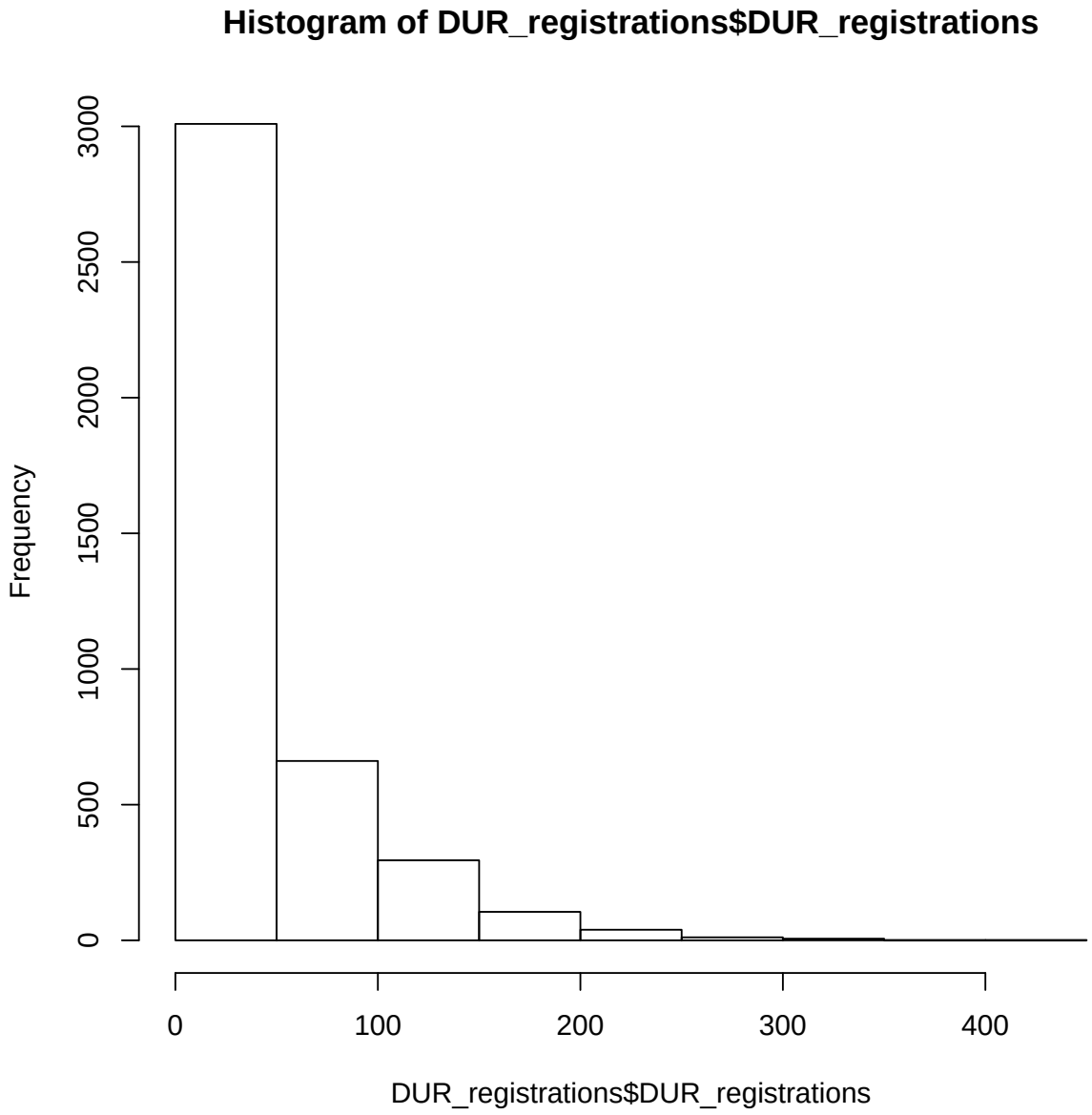

```
#   recency of registration

MAX_registration_date <-
  aggregate(registrationsIND[,"RegistrationDate",drop=FALSE],
            by=list(CompanyName=registrationsIND$CompanyName),
            max)

colnames(MAX_registration_date)[2] <- "MAX_registration_date"

str(MAX_registration_date, vec.len=1)

#-# 'data.frame': 4128 obs. of  2 variables:
```

```
#-#  $ CompanyName          : chr  "Action Academy" ...
#-#  $ MAX_registration_date: Date, format: "2014-07-02" ...

REC_registrations <-
  data.frame(CompanyName= MAX_registration_date$CompanyName,
             REC_registrations= as.numeric(t2) -
                    as.numeric(MAX_registration_date$MAX_registration_date),
             stringsAsFactors=FALSE)

summary(REC_registrations$REC_registrations)

#-#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#-#    0.00   10.00   23.00   41.03   54.00  393.00

hist(REC_registrations$REC_registrations)
```



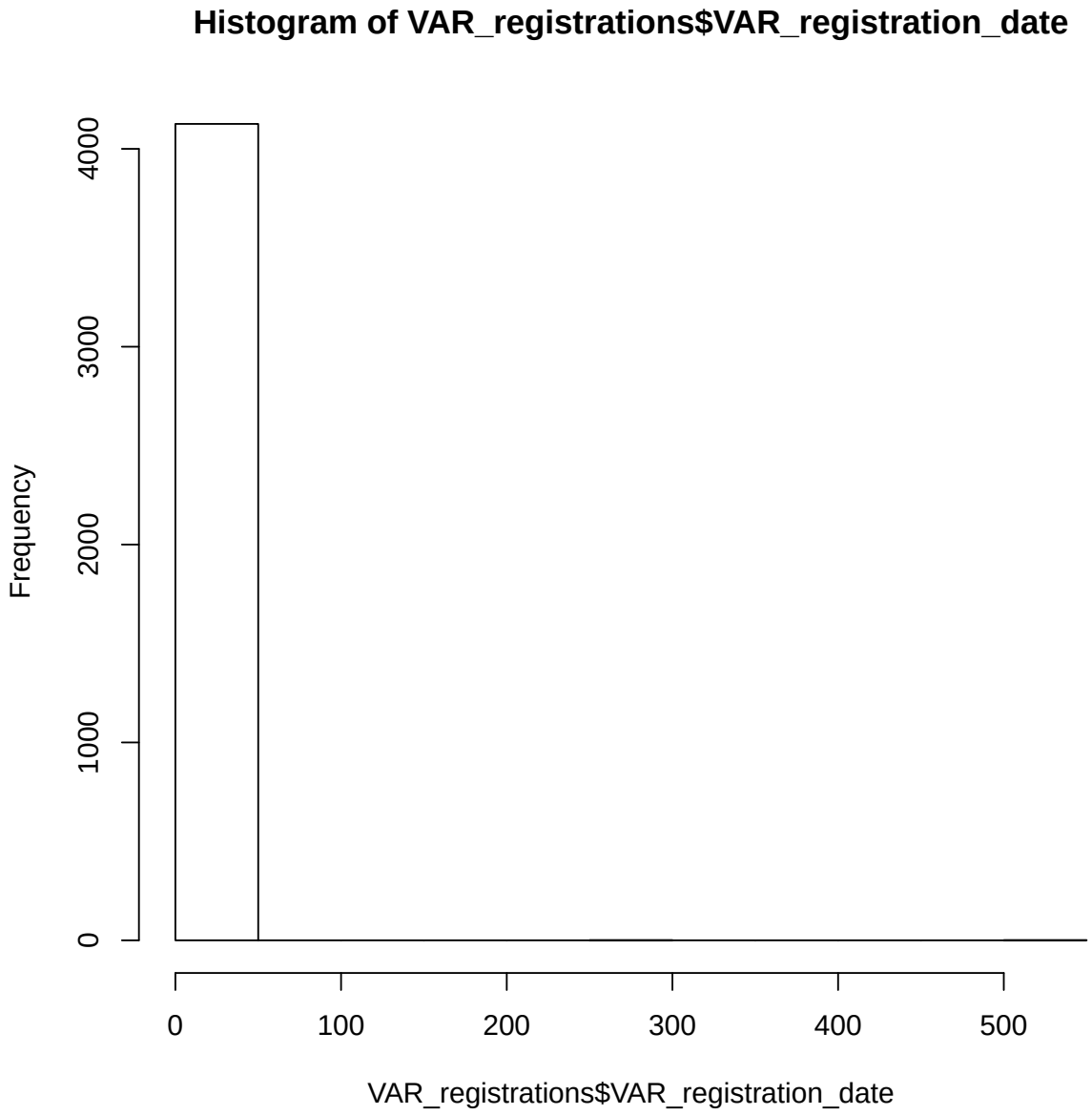

```
#   elapsed time since first registration

MIN_registration_date <-
  aggregate(registrationsIND[,"RegistrationDate",drop=FALSE],
            by=list(CompanyName=registrationsIND$CompanyName),
            min)

colnames(MIN_registration_date)[2] <- "MIN_registration_date"

str(MIN_registration_date, vec.len=1)

#-# 'data.frame': 4128 obs. of  2 variables:
```

```
#-#  $ CompanyName          : chr  "Action Academy" ...
#-#  $ MIN_registration_date: Date, format: "2014-05-31" ...

DUR_registrations <-
  data.frame(CompanyName=MIN_registration_date$CompanyName,
             DUR_registrations= as.numeric(t2) -
                   as.numeric(MIN_registration_date$MIN_registration_date),
             stringsAsFactors=FALSE)

str(DUR_registrations, vec.len=1)

#-# 'data.frame': 4128 obs. of  2 variables:
#-#  $ CompanyName      : chr  "Action Academy" ...
#-#  $ DUR_registrations: num  425 373 ...

summary(DUR_registrations$DUR_registrations)

#-#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#-#    0.00   10.00   23.00   41.09   54.00  425.00

hist(DUR_registrations$DUR_registrations)
```



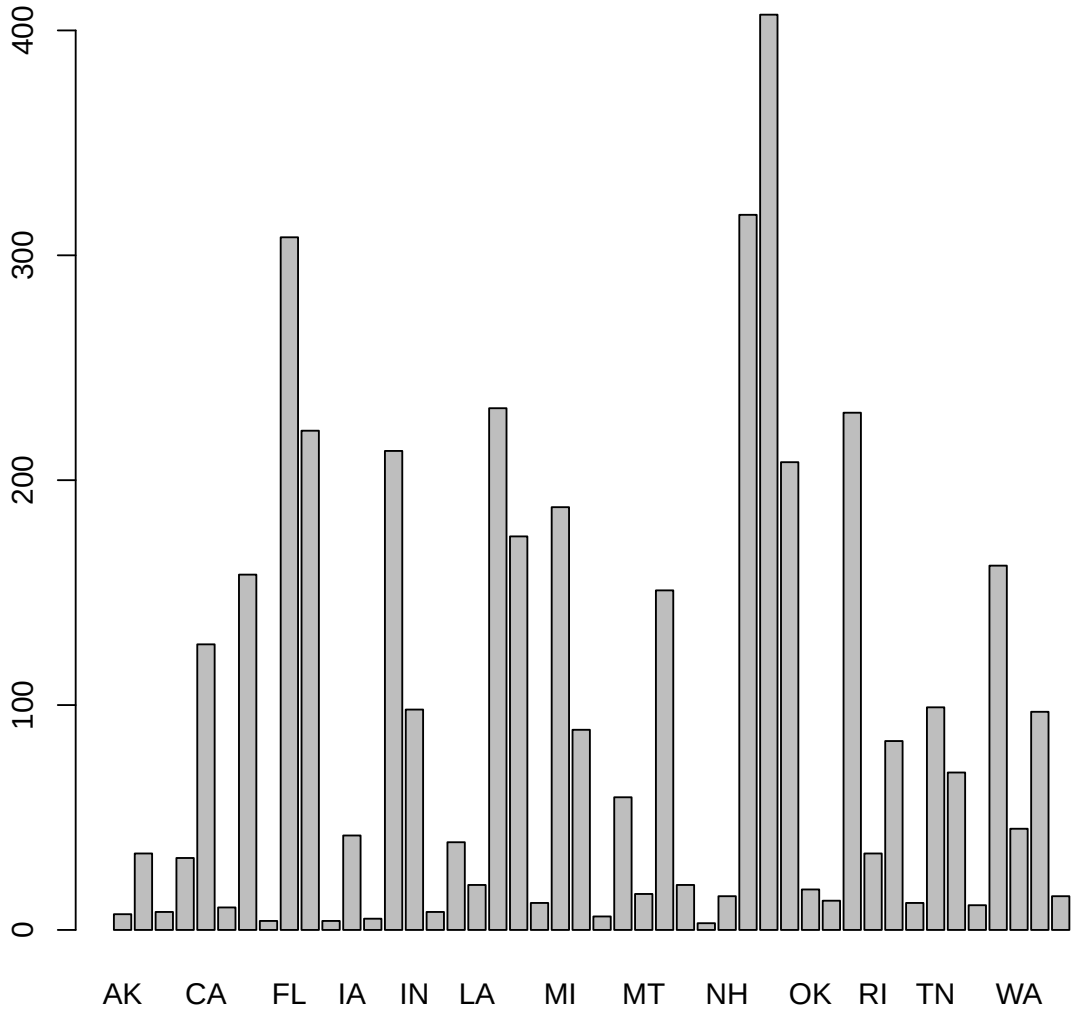

```
#   variance registration time

VAR_registrations <-
  aggregate(registrationsIND[,"RegistrationDate",drop=FALSE],
            by=list(CompanyName=registrationsIND$CompanyName),
            var)

colnames(VAR_registrations)[2] <- "VAR_registration_date"

NAs <- is.na(VAR_registrations$VAR_registration_date)
```

```
VAR_registrations$VAR_registration_date[NAs] <- 0

str(VAR_registrations, vec.len=1)

#-# 'data.frame': 4128 obs. of  2 variables:
#-#  $ CompanyName          : chr  "Action Academy" ...
#-#  $ VAR_registration_date: num  512 ...

summary(VAR_registrations$VAR_registration_date)

#-#     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
#-#   0.0000   0.0000   0.0000   0.2263   0.0000 512.0000

hist(VAR_registrations$VAR_registration_date)
```




```
#   state

#randomForest can handle categorical predictors
#with up to 53 categories. Since there are only
#50 states in the US we do not need to make
#dummy variables if we only use randomForest.

str(registrationsIND$CompanyAddress)

#-#  chr [1:10487] "1112 3rd Drive North, Woburn, MA 01801" ...

#remove duplicates
```

```
dups <- duplicated(registrationsIND[,c("CompanyName","CompanyAddress")])

registrationsIND <-
  registrationsIND[!dups, c("CompanyName","CompanyAddress")]

str(registrationsIND, vec.len=1)

#-# 'data.frame': 4128 obs. of  2 variables:
#-#  $ CompanyName   : chr  "Action Academy" ...
#-#  $ CompanyAddress: chr  "1112 3rd Drive North, Woburn, MA 01801" ...

pos <- unlist(gregexpr(",",registrationsIND$CompanyAddress))

pos <- pos[seq(2, length(pos), 2)]

state <- substr(registrationsIND$CompanyAddress,pos+2,pos+2+1)

state <- data.frame(registrationsIND[,"CompanyName"],
                    state,
                    stringsAsFactors=FALSE)

colnames(state)[1] <- "CompanyName"

str(state, vec.len=1)

#-# 'data.frame': 4128 obs. of  2 variables:
#-#  $ CompanyName: chr  "Action Academy" ...
#-#  $ state      : chr  "MA" ...

barplot(table(state$state))
```




```
# Merge independents and dependent
data <- list(state,
             VAR_registrations,
             DUR_registrations,
             REC_registrations,
             NBR_registrations,
             dependent)

str(data, vec.len=1)

#-# List of 6
```

```
#-#  $ :'data.frame': 4128 obs. of  2 variables:
#-#   ..$ CompanyName: chr [1:4128] "Action Academy" ...
#-#   ..$ state      : chr [1:4128] "MA" ...
#-#  $ :'data.frame': 4128 obs. of  2 variables:
#-#   ..$ CompanyName          : chr [1:4128] "Action Academy" ...
#-#   ..$ VAR_registration_date: num [1:4128] 512 ...
#-#  $ :'data.frame': 4128 obs. of  2 variables:
#-#   ..$ CompanyName      : chr [1:4128] "Action Academy" ...
#-#   ..$ DUR_registrations: num [1:4128] 425 373 ...
#-#  $ :'data.frame': 4128 obs. of  2 variables:
#-#   ..$ CompanyName      : chr [1:4128] "Action Academy" ...
#-#   ..$ REC_registrations: num [1:4128] 393 331 ...
#-#  $ :'data.frame': 4128 obs. of  2 variables:
#-#   ..$ CompanyName      : chr [1:4128] "Action Academy" ...
#-#   ..$ NBR_registrations: int [1:4128] 2 5 ...
#-#  $ :'data.frame': 4128 obs. of  2 variables:
#-#   ..$ CompanyName: chr [1:4128] "Action Academy" ...
#-#   ..$ Acquisition: Factor w/ 2 levels "0","1": 1 1 ...

# Look at the dimensions
# ?Number of instances should be identical
lapply(data,dim)

#-# [[1]]
#-# [1] 4128    2
#-#
#-# [[2]]
#-# [1] 4128    2
#-#
#-# [[3]]
#-# [1] 4128    2
#-#
#-# [[4]]
#-# [1] 4128    2
#-#
#-# [[5]]
#-# [1] 4128    2
#-#
#-# [[6]]
#-# [1] 4128    2

basetable <- Reduce(function(x,y) merge(x,y,by='CompanyName'),
                    data)

str(basetable, vec.len=1)

#-# 'data.frame': 4128 obs. of  7 variables:
#-#  $ CompanyName          : chr  "Action Academy" ...
#-#  $ state                : chr  "MA" ...
#-#  $ VAR_registration_date: num  512 ...
#-#  $ DUR_registrations    : num  425 373 ...
```

```
#-#  $ REC_registrations    : num  393 331 ...
#-#  $ NBR_registrations    : int  2 5 ...
#-#  $ Acquisition          : Factor w/ 2 levels "0","1": 1 1 ...

basetable$CompanyName <- NULL

Acquisition <- basetable$Acquisition

basetable$Acquisition <- NULL

str(basetable, vec.len=1)

#-# 'data.frame': 4128 obs. of  5 variables:
#-#  $ state                : chr  "MA" ...
#-#  $ VAR_registration_date: num  512 ...
#-#  $ DUR_registrations    : num  425 373 ...
#-#  $ REC_registrations    : num  393 331 ...
#-#  $ NBR_registrations    : int  2 5 ...

#Any missing values?
colSums(is.na(basetable))

#-#                 state VAR_registration_date
#-#                     0                     0
#-#     DUR_registrations     REC_registrations
#-#                     0                     0
#-#     NBR_registrations
#-#                     0

sum(is.na(Acquisition))

#-# [1] 0

#Change state to factor for the modeling phase
basetable$state <- as.factor(basetable$state)

#Our basetable is now ready and we can
#move on the modeling phase
```

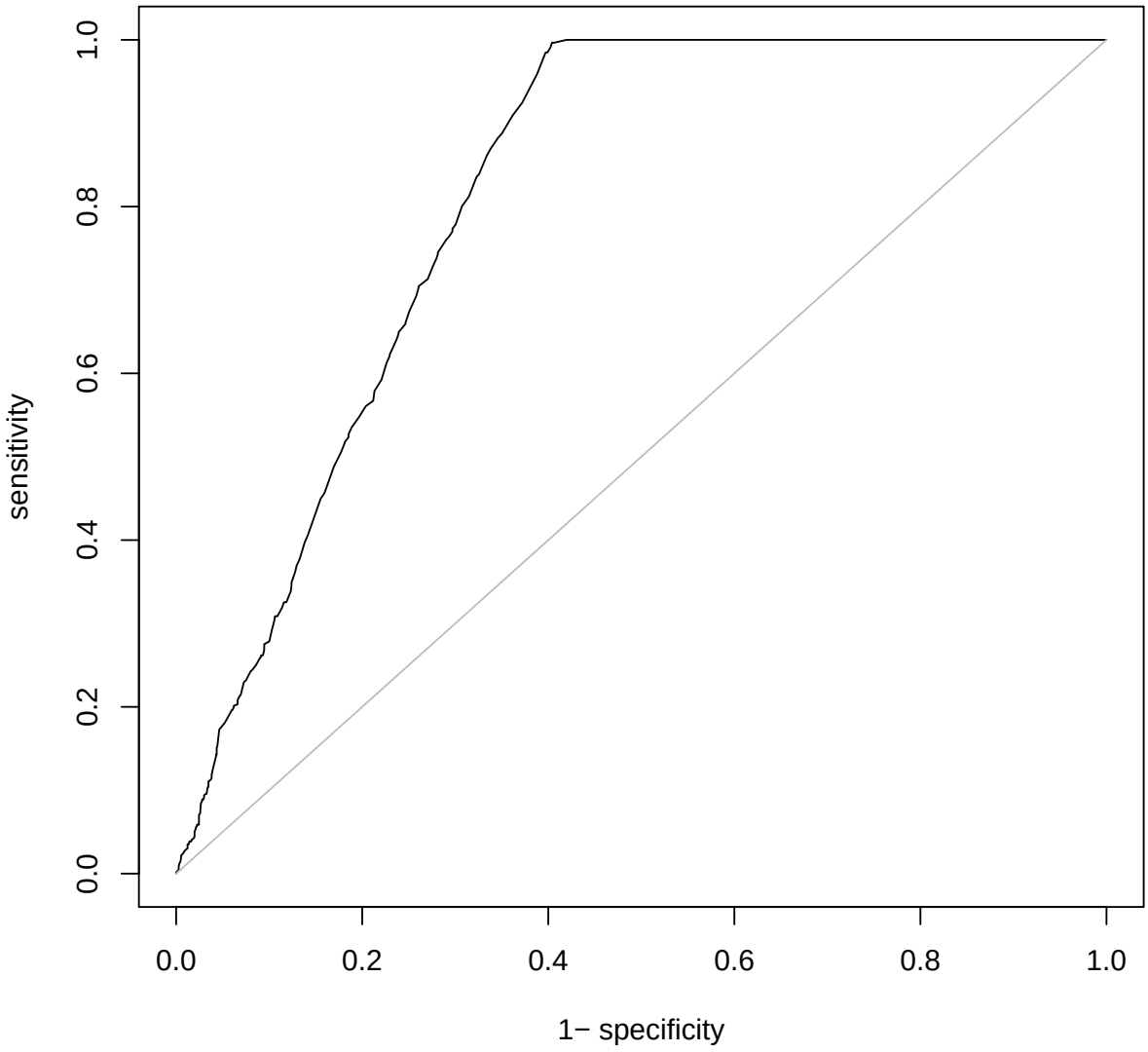
3.4 Modeling

```
#We'll be using random forest and hence we will
#not require a validation set.
#Split the data in a train and test set
ind <- 1:nrow(basetable)
indTRAIN <- sample(ind,round(0.5*length(ind)))
indTEST <- ind[-indTRAIN]
```

```
#Load the randomForest package
if (!require('randomForest')){
  install.packages('randomForest',
                   repos='http://cran.rstudio.com',
                   quiet=TRUE)
  library("randomForest")
}

#Fit the random forest on the training set
rf <- randomForest(x=basetable[indTRAIN,],
                   y=Acquisition[indTRAIN])

#Deploy the forest on the test set
pred <- predict(rf,basetable[indTEST,],
                type="prob")[,2]

#Next we need to evaluate our model. If our model
#perfomance is not satisfactory we might need
#to go back to data modeling or even data
#preparation.
```

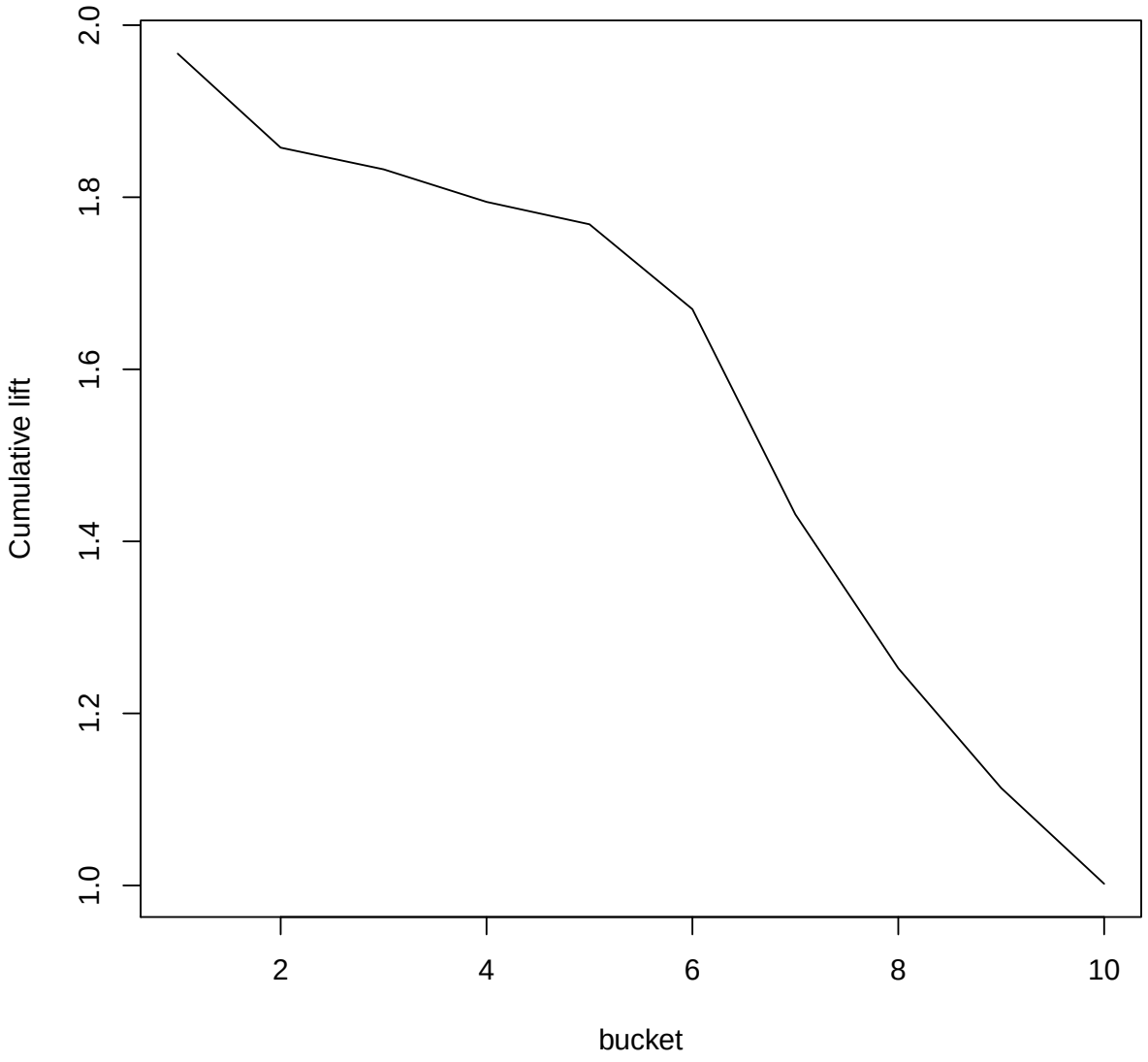
3.5 Model Evaluation

```
#Load the AUC package
if (!require('AUC')){
  install.packages('AUC',
                   repos='http://cran.rstudio.com',
                   quiet=TRUE)
  library("AUC")
}

#Plot the ROC curve
plot(roc(pred,Acquisition[indTEST]))
```



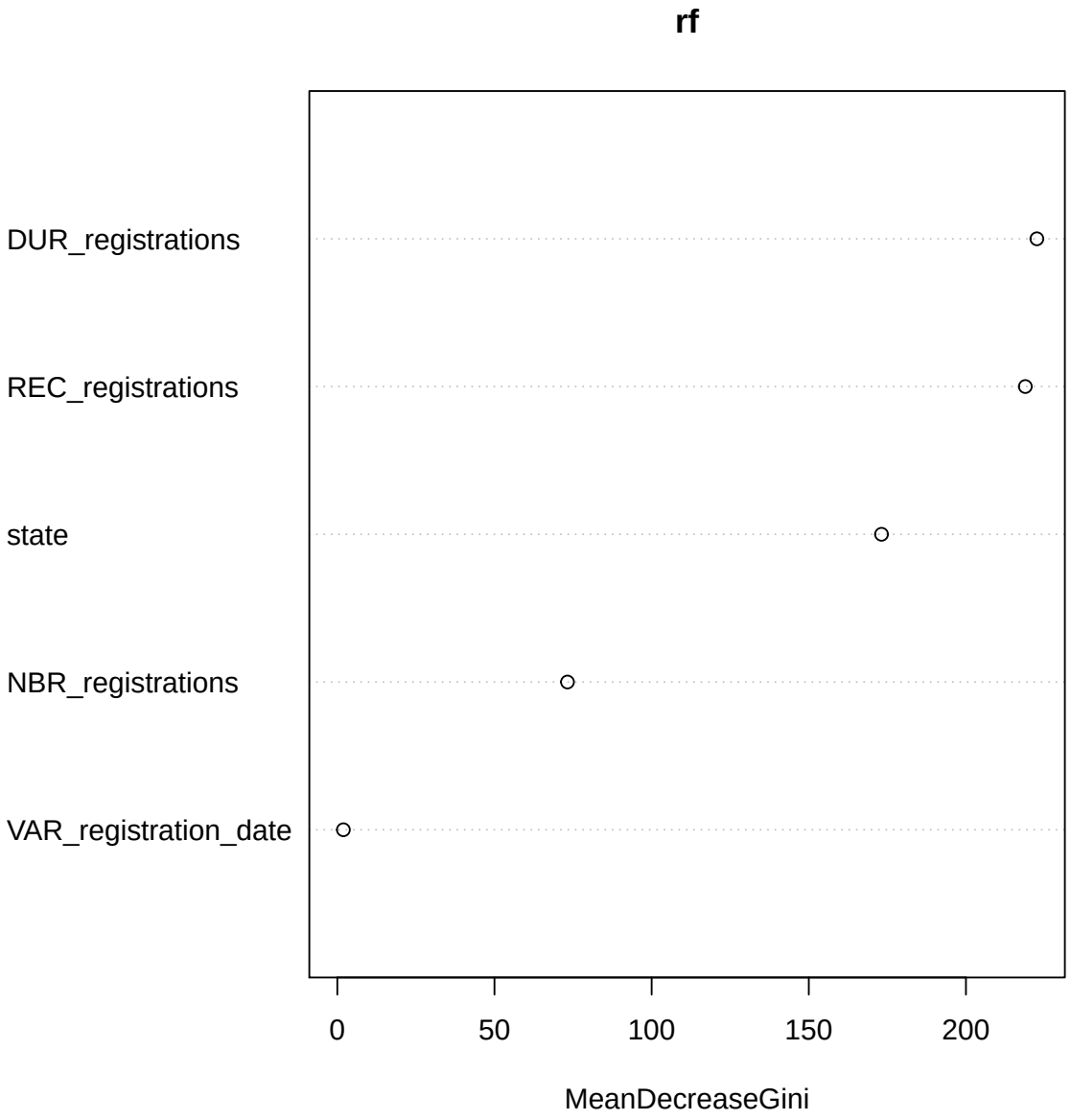

```
#Compute the AUC
auc(roc(pred,Acquisition[indTEST]))

#-# Error in rank(prob): argument "prob" is missing, with no default

#This is a very good AUC

#Load the lift package
if (!require('lift')){
  install.packages('lift',
                   repos='http://cran.rstudio.com',
```

```
                    quiet=TRUE)
  library('lift')
}


#Plot the lift curve
plotLift(pred,Acquisition[indTEST])
```



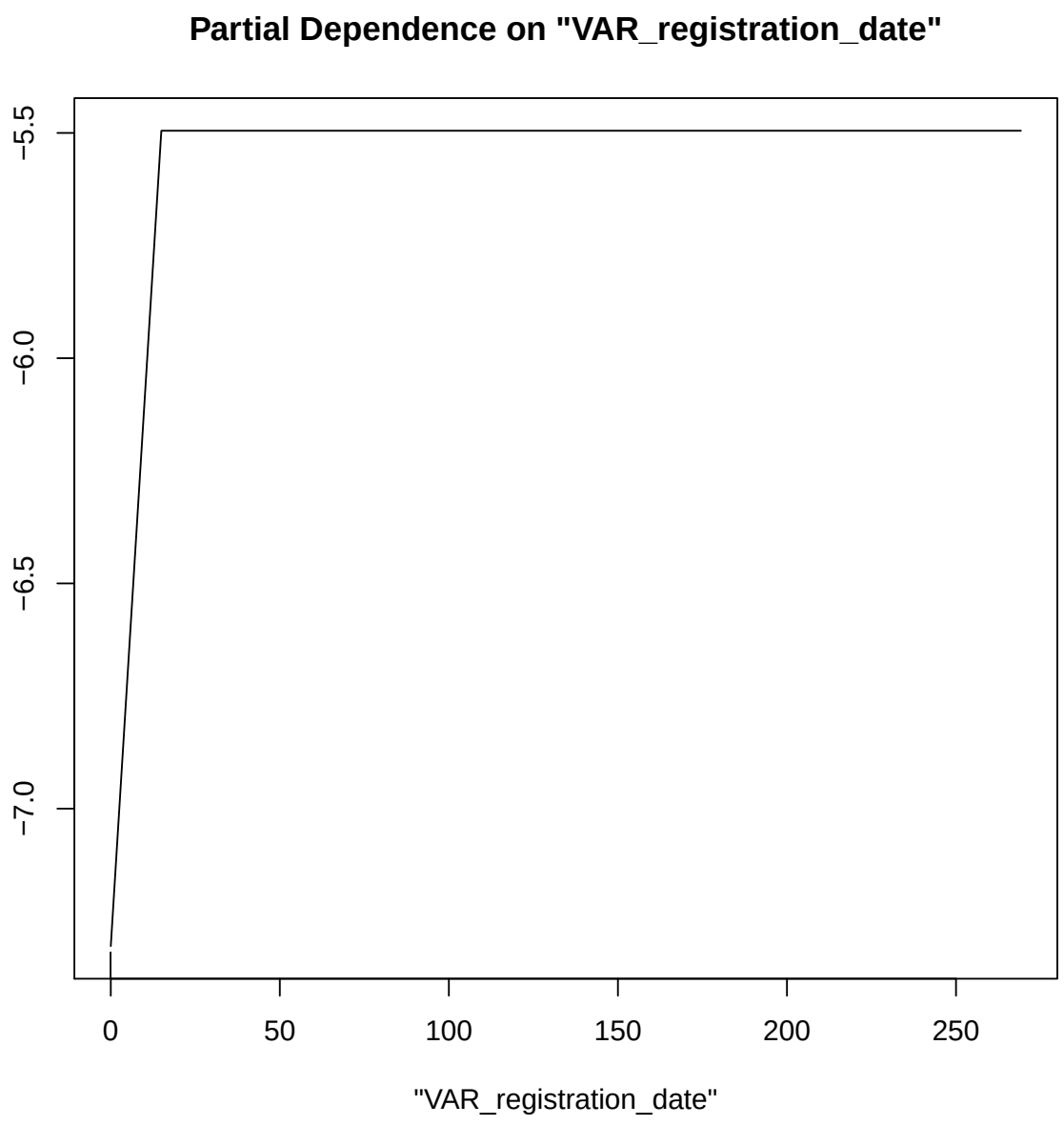

```
#Compute the top decile lift
TopDecileLift(pred,Acquisition[indTEST])
```

```
#-# [1] 1.967

#We want to have some insight into our model
#and make sure the relationships are plausible
#The first step is to look at which variables
#are important
varImpPlot(rf)
```

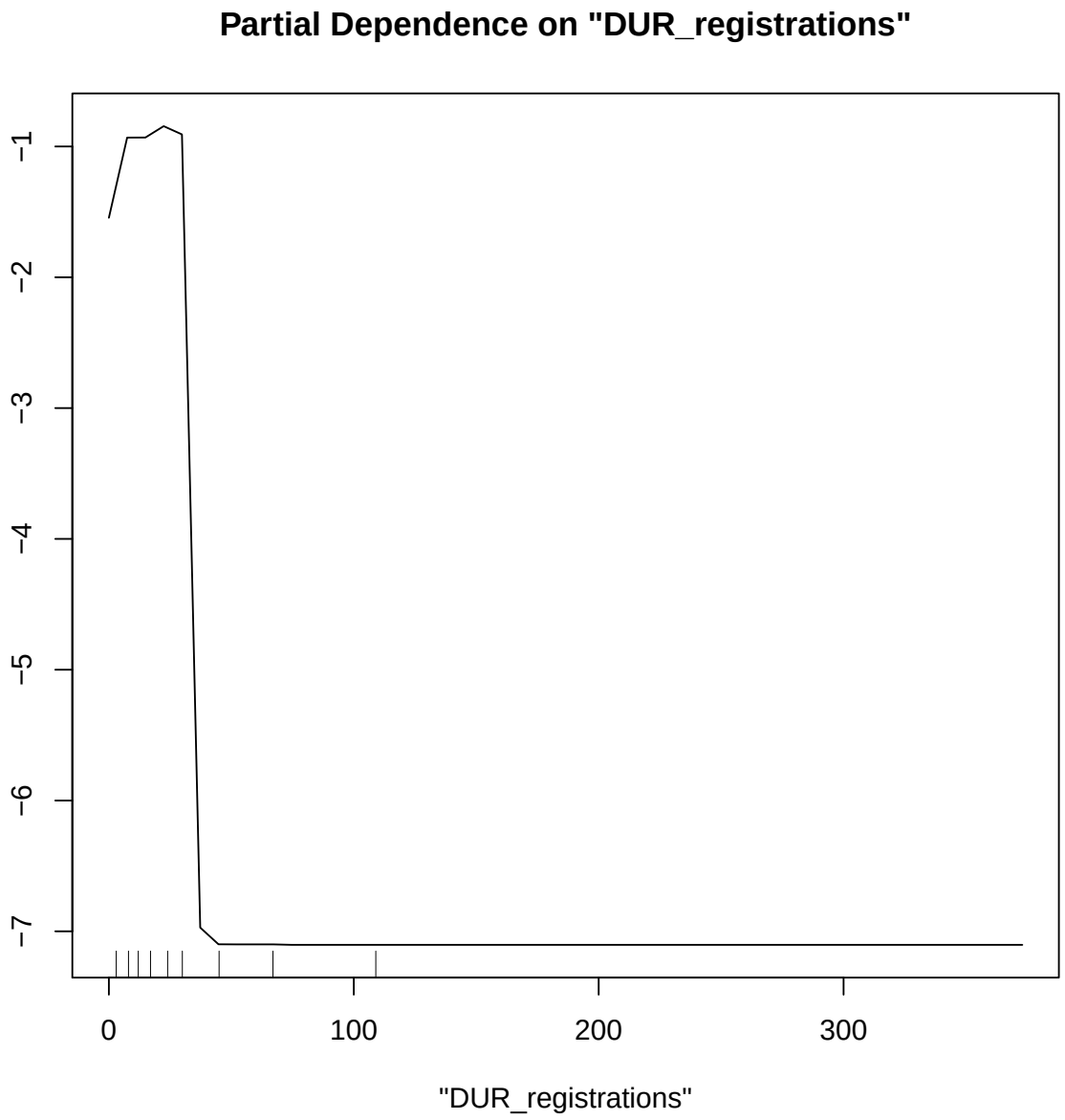



```
#The second step is to look at some of the relationships

partialPlot(x=rf,x.var="VAR_registration_date",
```

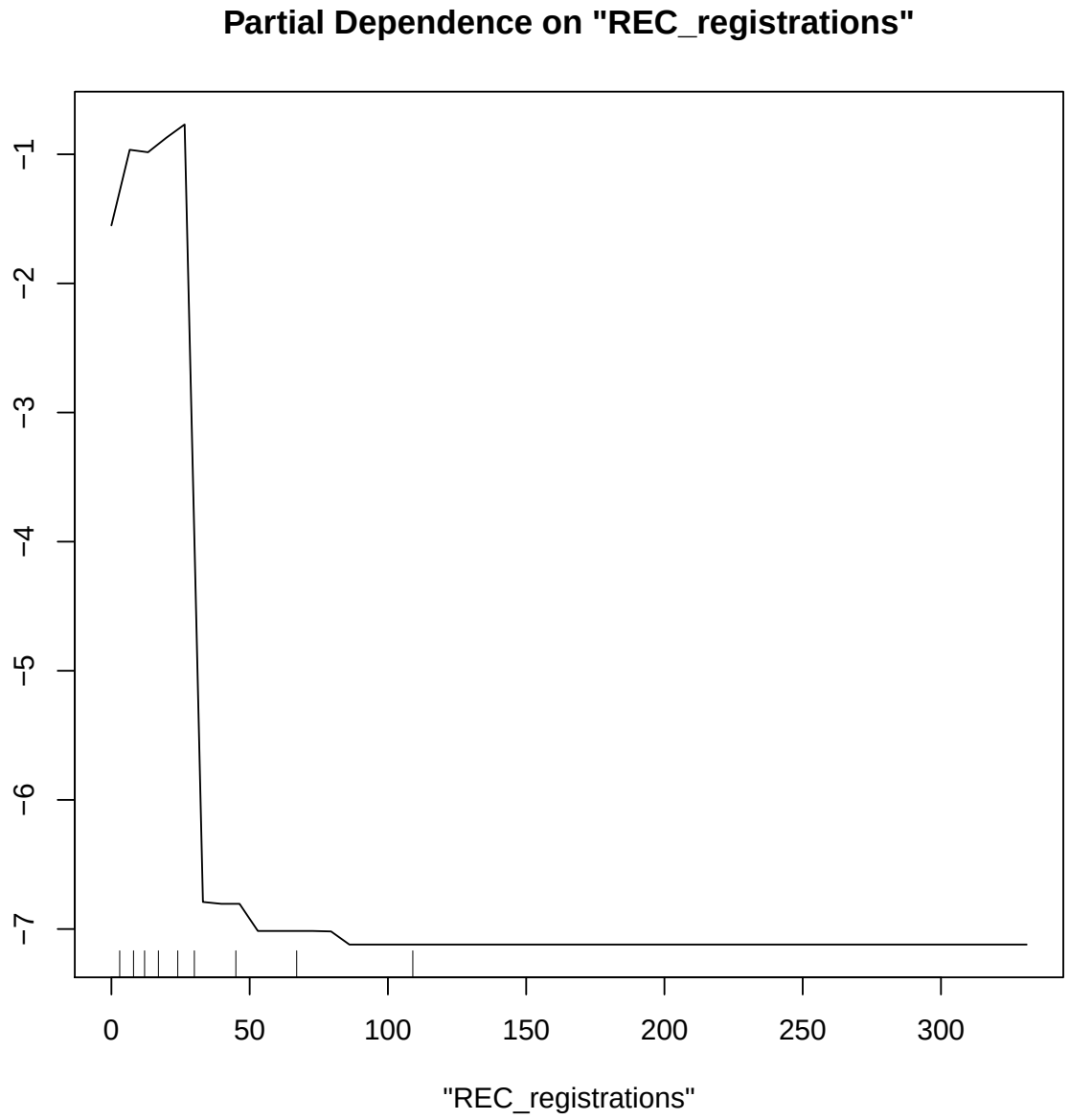
```
            pred.data=basetable[indTEST,],which.class=1)
```



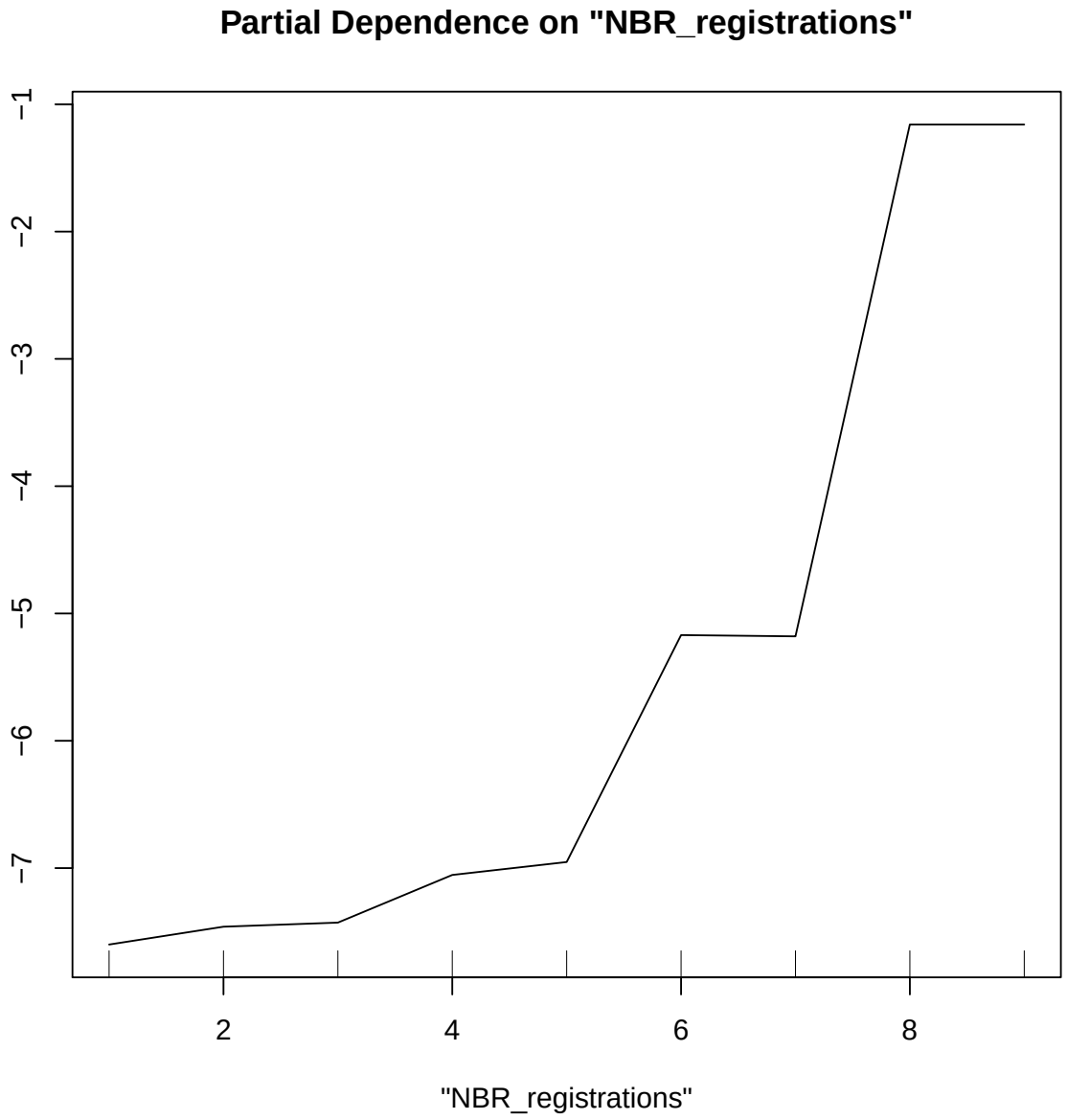

```
#This plot tells us that more variance in the
#registration date results in a higher
#propensity to be acquired

partialPlot(x=rf,x.var="DUR_registrations",
            pred.data=basetable[indTEST,],which.class=1)
```




```
#The more time has elapsed since the first registration
#the lower the propensity to be acquired

partialPlot(x=rf,x.var="REC_registrations",
            pred.data=basetable[indTEST,],which.class=1)
```




```
#The more time has elapsed since the last registration
#the lower the propensity to be acquired

partialPlot(x=rf,x.var="NBR_registrations",
            pred.data=basetable[indTEST,],which.class=1)
```




```
#More registrations results in a higher propensity
#to be acquired

#All the relationships are plausible and intuitive.
#In addition the lift and AUC are very good.
#Hence we can conclude the modeling phase
```

3.6 Model Deployment

3.6.1 Function to create the model

```
rm(list=ls())

# acquireR reads in the data, prepares the predictors and
# the dependent variable, evaluates the model and creates
# the final model.

# There are six parameters:
# start_ind Character string. The start of the independent period ("%m/%d/%Y")
# end_ind Character string. The end of the independent period ("%m/%d/%Y")
# start_dep Character string. The start of the dependent period ("%m/%d/%Y")
# end_dep Character string. The end of the dependent period ("%m/%d/%Y")
# evaluate Boolean. Should the model be evaluated?

# An overview of the function using pseudo code:
# acquireR <- function(start_ind,
#                      end_ind,
#                      start_dep,
#                      end_dep,
#                      evaluate=TRUE){
#
#
#   Load required packages
#
#   Create function readAndPrepareData as follows
#   readAndPrepareData(train=TRUE,...){
#     read in data
#     create predictors
#     if train==TRUE return predictors and dependent, otherwise predictors
#   }
#
#   Call readAndPrepareData
#
#   If evaluate==TRUE create train and test and evaluate model
#
#   Create model on all data
#
#   Store in a list:
#       the model
#       the readAndPrepareData function
#       the format of the dates
#       the length of the independent period
#
#   Set that list to class "acquireR"
#
#   Return that list
# }
```

```
acquireR <- function(start_ind,
                     end_ind,
                     start_dep,
                     end_dep,
                     evaluate=TRUE) {

f <- "%m/%d/%Y"
t1 <- as.Date(start_ind, f)
t2 <- as.Date(end_ind, f)
t3 <- as.Date(start_dep, f)
t4 <- as.Date(end_dep, f) #dump date
length_ind <- t2 - t1

#Load all required packages
for (i in c("AUC","lift","randomForest")) {
  if (!require(i,character.only=TRUE,quietly=TRUE)) {
    install.packages(i,
                     repos='http://cran.rstudio.com',
                     quiet=TRUE)
    require(i,
            character.only=TRUE,
            quietly=TRUE)
  }
}


readAndPrepareData <- function(train=TRUE,...){


  #DATA UNDERSTANDING
  ###################################################################

  cat("Reading in data:")
  time <- Sys.time()

  #We can read the data using the read.csv function
  customers <-
    read.csv("http://ballings.co/hidden/aCRM/data/chapter3/customers.csv",
             header=TRUE,
             colClasses="character")

  purchases <-
    read.csv("http://ballings.co/hidden/aCRM/data/chapter3/purchases.csv",
             header=TRUE,
             colClasses=c("character",
                          "Date",
                          "character",
                          "numeric"))

  registrations <-
    read.csv("http://ballings.co/hidden/aCRM/data/chapter3/registrations.csv",
```

```
            header=TRUE,
            colClasses=c("character",
                         "character",
                         "character",
                         "character",
                         "Date"))

if (train==TRUE){
  #To avoid error message in predict:
  #New factor levels not present in the training data
  pos <- unlist(gregexpr(",",registrations$CompanyAddress))
  pos <- pos[seq(2, length(pos), 2)]
  state <- substr(registrations$CompanyAddress,pos+2,pos+2+1)
  levelsState <- levels(as.factor(state))
}

cat(format(round(as.numeric(Sys.time()- time),1),nsmall=1,width=4),
    attr(Sys.time()- time,"units"), "\n")

cat("Preparing basetable:")
time <- Sys.time()
#################################################################
#DATA PREPARATION

#Because they have a 1:1 relationship, we can merge
#purchases and customers
pur_cus <- merge(purchases,customers,by="CustomerID")

# When we are calling readAndPrepareData in predict:
if (train==FALSE){
      dots <- list(...) # list(...) evaluates all arguments and
                        # returns them in a named list
      t2 <- dots$end_ind
      t1 <- t2 - dots$length_ind

      rm(dots)
}
#Otherwise just fetch timewindow from
#surrounding environment

#Split data into independent and dependent data
registrationsIND <- registrations[registrations$RegistrationDate <= t2 &
                    registrations$RegistrationDate >= t1,]

if (train==TRUE){
pur_cusDEP <- pur_cus[pur_cus$PurchaseDate > t3 &
                      pur_cus$PurchaseDate <= t4,]
}

#Make sure to only select companies that have not purchased before t2
pur_cus_bef_t2 <- unique(pur_cus[pur_cus$PurchaseDate <= t2,"CompanyName"])
```

```
registrationsIND <-
  registrationsIND[!registrationsIND$CompanyName %in%
                   pur_cus_bef_t2,]

if (train==TRUE){
  #Compute dependent
  #This comes down to merging registrationsIND with pur_cusDEP
  #Note the left outer merge
  dependent <-
    merge(data.frame(CompanyName=unique(registrationsIND[,"CompanyName"]),
                     stringsAsFactors=FALSE),
          data.frame(pur_cusDEP[,"CompanyName",drop=FALSE],Acquisition=1),
          by="CompanyName", all.x=TRUE)

  dependent$Acquisition[is.na(dependent$Acquisition)] <- 0

  dependent$Acquisition <- as.factor(dependent$Acquisition)

}

#Compute predictor variables
#We can only use the registrationsIND table

#   Number of registrations

NBR_registrations <- aggregate(registrationsIND[,"CompanyName",drop=FALSE],
                               by=list(CompanyName=registrationsIND$CompanyName),
                               length)

colnames(NBR_registrations)[2] <- "NBR_registrations"


#   recency of registration

MAX_registration_date <-
  aggregate(registrationsIND[,"RegistrationDate",drop=FALSE],
            by=list(CompanyName=registrationsIND$CompanyName),
            max)

colnames(MAX_registration_date)[2] <- "MAX_registration_date"


REC_registrations <-
  data.frame(CompanyName= MAX_registration_date$CompanyName,
             REC_registrations= as.numeric(t2) -
                    as.numeric(MAX_registration_date$MAX_registration_date),
             stringsAsFactors=FALSE)


#   elapsed time since first registration
```

```
MIN_registration_date <-
  aggregate(registrationsIND[,"RegistrationDate",drop=FALSE],
            by=list(CompanyName=registrationsIND$CompanyName),
            min)

colnames(MIN_registration_date)[2] <- "MIN_registration_date"


DUR_registrations <-
  data.frame(CompanyName=MIN_registration_date$CompanyName,
             DUR_registrations= as.numeric(t2) -
                   as.numeric(MIN_registration_date$MIN_registration_date),
             stringsAsFactors=FALSE)


#   variance registration time

VAR_registrations <-
  aggregate(registrationsIND[,"RegistrationDate",drop=FALSE],
            by=list(CompanyName=registrationsIND$CompanyName),
            var)

colnames(VAR_registrations)[2] <- "VAR_registration_date"

NAs <- is.na(VAR_registrations$VAR_registration_date)

VAR_registrations$VAR_registration_date[NAs] <- 0


#   state

#remove duplicates
dups <- duplicated(registrationsIND[,c("CompanyName","CompanyAddress")])

registrationsIND <-
  registrationsIND[!dups, c("CompanyName","CompanyAddress")]



pos <- unlist(gregexpr(",",registrationsIND$CompanyAddress))

pos <- pos[seq(2, length(pos), 2)]

state <- substr(registrationsIND$CompanyAddress,pos+2,pos+2+1)

state <- data.frame(registrationsIND[,"CompanyName"],
                    state,
                    stringsAsFactors=FALSE)

#Change state to factor for the modeling phase
state$state <- as.factor(state$state)
```

```
  if (train==TRUE){
    #To avoid error in predict:
    #New factor levels not present in the training data
    levels(state$state) <- levelsState
  }

  colnames(state)[1] <- "CompanyName"


  # Merge independents and dependent
  if (train==TRUE){
  data <- list(state,
               VAR_registrations,
               DUR_registrations,
               REC_registrations,
               NBR_registrations,
               dependent)
  } else {
  data <- list(state,
               VAR_registrations,
               DUR_registrations,
               REC_registrations,
               NBR_registrations)
  }


  basetable <- Reduce(function(x,y) merge(x,y,by='CompanyName'),
                      data)


  if (train==TRUE){
    basetable$CompanyName <- NULL
    Acquisition <- basetable$Acquisition
    basetable$Acquisition <- NULL
  }

  cat(format(round(as.numeric(Sys.time()- time),1),nsmall=1,width=4),
      attr(Sys.time()- time,"units"), "\n")

  #Our basetable is now ready and we can
  #move on the modeling phase
  if (train==TRUE){
    return(list(predictors=basetable, Acquisition=Acquisition))
  } else {
    return(basetable)
  }
}#end readAndPrepareData

basetable <- readAndPrepareData()

#####################################################################
```

```
#MODELING


if (evaluate==TRUE){
      cat("Evaluating model:")
      time <- Sys.time()

      #We'll be using random forest and hence we will
      #not require a validation set.
      #Split the data in a train and test set
      ind <- 1:nrow(basetable$predictors)
      indTRAIN <- sample(ind,round(0.5*length(ind)))
      indTEST <- ind[-indTRAIN]

      #Fit the random forest on the training set
      rf <- randomForest(x=basetable$predictors[indTRAIN,],
                  y=basetable$Acquisition[indTRAIN])

      #Deploy the forest on the test set
      pred <- predict(rf,basetable$predictors[indTEST,],
                type="prob")[,2]

      cat(format(round(as.numeric(Sys.time()- time),1),nsmall=1,width=4),
      attr(Sys.time()- time,"units"), "\n")

      cat("   Number of predictors:", ncol(basetable$predictors[indTRAIN,]),
          "predictors\n")
      cat("   AUROC:",
           round(auc(roc(pred,basetable$Acquisition[indTEST])),4),"\n")
      cat("   Top decile lift of:",
           round(TopDecileLift(pred,basetable$Acquisition[indTEST]),4),"\n")
}

cat("Creating model:")
time <- Sys.time()

#Build model on all data
rf <- randomForest(x=basetable$predictors,
                   y=basetable$Acquisition)

cat(format(round(as.numeric(Sys.time()- time),1),nsmall=1,width=4),
      attr(Sys.time()- time,"units"), "\n")
l <- list(rf = rf,
          readAndPrepareData = readAndPrepareData,
          f = f,
          length_ind = length_ind)
class(l) <- "acquireR"
return(l)
}
```

```
acquisitionModel <-
   acquireR(start_ind="05/31/2014",
            end_ind="07/30/2015",
            start_dep="07/31/2015",
            end_dep="08/30/2015",
            evaluate=TRUE)

#-# Reading in data: 3.6 secs
#-# Preparing basetable: 0.3 secs
#-# Evaluating model: 0.8 secs
#-#    Number of predictors: 5 predictors

#-# Error in rank(prob): argument "prob" is missing, with no default
```

3.6.2 Function to create predictions

```
# predict.acquireR reads in the data, prepares the predictors and
# creates the predictions

# There are three parameters:
# object The output of the acquireR function
# dumpDate Character string. The date that the data was extracted
#                            from the database ("%m/%d/%Y")
# Overview of predict.acquire using pseudo code:
# predict.acquireR <- function(object, dumpDate) {
#
#   Load required packages
#
#   Call object$readAndPrepareData function with train=FALSE
#   to create predictors
#
#   Predict
#
#   Store company name and predictions in data frame
#
#   Sort by predictions
#
#   Return the data frame
# }

predict.acquireR <- function(object, dumpDate) {

#Load all required packages
for (i in c("AUC","lift","randomForest")) {
  if (!require(i,character.only=TRUE,quietly=TRUE)) {
    install.packages(i,
                     repos='http://cran.rstudio.com',
                     quiet=TRUE)
```

```
    require(i,
            character.only=TRUE,
            quietly=TRUE)
  }
}

#Make sure all variables in the readAndPrepareData
#enclosing environment (where it was defined) are removed
#to avoid unexpected results
environment(object$readAndPrepareData) <- environment()


basetable <- object$readAndPrepareData(train=FALSE,
                                       end_ind= as.Date(dumpDate, object$f),
                                       length_ind=object$length_ind)

cat("Predicting: ")
time <- Sys.time()

ans <- data.frame(CompanyName=basetable$CompanyName,
                  Score=predict(object=object$rf,
                                newdata=basetable,
                                type="prob")[,2])
ans <- ans[order(ans$Score, decreasing=TRUE),]

cat(format(round(as.numeric(Sys.time()- time),1),nsmall=1,width=4),
      attr(Sys.time()- time,"units"), "\n")
ans

}

pred <-
  predict(object=acquisitionModel,
          dumpDate="08/30/2015")

#-# Error in predict(object = acquisitionModel, dumpDate = "08/30/2015"): object 'acquisitionModel'
not found

head(pred)

#-# Error in head(pred): object 'pred' not found
```

4

Case study: Customer Up-sell

5

Case study: Customer Cross-sell

Deliverables:

- ERD
- Code flowcharct
- One function with parameters start_ind, end_ind, start_dep, end_dep, evaluate, and next (all or first), containing all the code to create the model .
- One function with parameters object, and dumpDate, containing all the code the make predictions using the model and new data

The quality of the two functions depends on performance (75%), code maintainability (shorter code is better)(15%), and run time (faster is better)(10%).

5.1 Business Understanding

The HVC database contains information about the transactions of a Home Vending Company (ice cream). The company has several sales representatives that sell its products at the customers' homes. Each customer is assigned to a specific route. Each of these routes will be completed by a sales representative in a biweekly schedule. The HVC wants two recommendation engines: (1) which product will a customer buy next (next product to buy model) in the dependent period, and (2) all the products a customer will buy in the dependent period.

5.2 Data Understanding

```
# http://kddata.co/data/chapter6/routes.csv
# http://kddata.co/data/chapter6/products.csv
# http://kddata.co/data/chapter6/customers.csv
# http://kddata.co/data/chapter6/visitdetails.csv
# http://kddata.co/data/chapter6/visits.csv
```

5.3 Data Preparation

5.4 Modeling

5.5 Model Evaluation

5.6 Model Deployment

6

Case study: Customer Defection

Deliverables:

- ERD
- Code flowcharct
- One function with parameters start_ind, end_ind, start_dep, end_dep, and evaluate, containing all the code to create the model .
- One function with parameters object, and dumpDate, containing all the code the make predictions using the model and new data

The quality of the two functions depends on performance (top decile lift and AUC)(75%), code maintainability (shorter code is better)(15%), and run time (faster is better)(10%).

6.1 Business Understanding

A newspaper publishing company (NPC) has been facing increasing churn rates since years. This evolution has been jumpstarted by the rise of news- websites, and has continued at an accelerating rate due to the popularity of tablet computers and (social) news- aggregator applications. Hence the NPC requires a predictive churn model in order to predict which customers will not renew their newspaper subscriptions. Customers are not allowed to cancel their subscriptions, and therefore a customer is defined as a churner if he or she does not renew the subscription once the current subscriptions ends.

Our job is to build a defection model to predict which customers are most likely to churn. We can then rank the customers by their churn propensity and the NPC can then contact those customers most likely to churn in order to try to retain them.

6.2 Data Understanding

The NPC has given us a very clear data dictionary for each of the tables. There are six tables: Customers, Subscriptions, Delivery, Formula, Credit, Complaints. Two important things are not immediately clear from looking at the table descriptions. First of all, the Formula table contains data about whether the subscription came from a campaign. Second, the Credit table describes credit that the customer might receive due to operational problems from the NPC.

Table 6.1: Data description of the Customers table

Variable name	Description
CustomerID	Unique customer identifier
Gender	M=Male F=Female NA=Not Available
DoB	Date of Birth
District	Macro geographical grouping: District 1-8
ZIP	Meso geographical grouping
StreetID	Micro geographical grouping: Street identifiers

Table 6.2: Data description of the Formula table

Variable name	Description
FormulaID	Formula identifier.
FormulaCode	Formula code
FormulaType	CAM=Campaign REG=Regular
Duration	Duration of the formula in months.

Table 6.3: Data description of the Subscriptions table

Variable name	Description
SubscriptionID	Unique subscription identifier
CustomerID	Customer identifier
ProductID	Product identifier
Pattern	Denotes delivery days. The position in the string equals the days in the week {1=delivery, 0=non-delivery}. For example: 1000000= delivery only on Monday 1111110= deliver on all days (except Sundays)
StartDate	The subscription's start date
EndDate	The subscription's end date
NbrNewspapers	Total number of copies, including NbrStart as determined at the start of the subscription.
NbrStart	Maximum number of copies before payment is received.
RenewalDate	Date that renewal is processed.
PaymentType	BT= Bank transfer DD= Direct debit
PaymentStatus	{Paid, Not Paid}
PaymentDate	Date of payment
FormulaID	Formula identifier.
GrossFormulaPrice	GrossFormulaPrice = NetFormulaPrice + TotalDiscount
NetFormulaPrice	NetFormulaPrice = GrossFormulaPrice - TotalDiscount
NetNewspaperPrice	Price of a single newspaper
ProductDiscount	Discount based on the product
FormulaDiscount	Discount based on the formula
TotalDiscount	TotalDiscount= ProductDiscount + FormulaDiscount
TotalPrice	TotalPrice = NbrNewspapers * NetNewspaperPrice TotalPrice = NetFormulaPrice + TotalCredit
TotalCredit	The customer's credit of previous transactions. Credit is negative, debit is positive.

Table 6.4: Data description of the Delivery table

Variable name	Description
DeliveryID	Unique delivery identifier
SubscriptionID	Unique subscription identifier
DeliveryType	DR=Delivery Rerouting MD=Main Delivery DI=Delivery Interruption
DeliveryClass	NOR= Normal ABN=Abnormal: every delivery that deviates from normal delivery (see DeliveryContext)
DeliveryContext	OTH= Other ACH= Address Change PCH= Product Change REN= Renewal NPA= Non- Payment VAC= Vacation
StartDate	Start date delivery.
EndDate	End date delivery.

Table 6.5: Data description of the Complaints table

Variable name	Description
ComplaintID	Unique complaint identifier
CustomerID	Customer identifier
ProductID	Product Identifier
ComplaintDate	Date of complaint
ComplaintType	1= Non delivery
	2= Late delivery
	3= Wrong product
	4= Messy delivery
	5= No post delivery
	6= Delivery while nothing ordered
	7= Damaged product
	8= Wrong address
	9= Other
SolutionType	1= Assign Credit
	2= Post delivery
	3= No solution needed
	4= Take precautions
FeedbackType	1= Force Majeure
	2= Weather conditions
	3= Error postman
	4= Late delivery to postal office
	5= Postman confirmed delivery
	6= Strike postal office

Table 6.6: Data description of the Credit table

Variable name	Description
CreditID	Unique credit identifier
SubscriptionID	Unique subscription identifier
ActionType	PO= Payout NC= No consequence EN= Extra Newspapers CC= Create Credit
ProcessingDate	Date on which the credit is processed.
CreditSource	SUB= Subscription TRA= Transaction (e.g, Double payment) COM= Complaint
Amount	Credit Amount in $
NbrNewspapers	Number of newspapers as credit. Only relevant when ActionType=EN

```
http://ballings.co/hidden/aCRM/data/chapter6/complaints.txt
http://ballings.co/hidden/aCRM/data/chapter6/credit.txt
http://ballings.co/hidden/aCRM/data/chapter6/customers.txt
http://ballings.co/hidden/aCRM/data/chapter6/delivery.txt
http://ballings.co/hidden/aCRM/data/chapter6/formula.txt
http://ballings.co/hidden/aCRM/data/chapter6/subscriptions.txt
```

6.3 Data Preparation

6.4 Modeling

6.5 Model Evaluation

6.6 Model Deployment

7

Case study: Customer Lifetime Value

Deliverables:

- Data dictionary
- ERD
- Code flowchart
- One function with parameters start_ind, end_ind, start_dep, end_dep, and evaluate, containing all the code to create the model.
- One function with parameters object, and dumpDate, containing all the code the make predictions using the model and new data

The quality of the two functions depends on performance (top decile lift and AUC)(75%), code maintainability (shorter code is better)(15%), and run time (faster is better)(10%).

In addition to capturing whether a customer will yes or no buy, a Customer Lifetime Value (CLV) model, has the advantage that it predicts the future value of the customer. In a sense it is thus less specific, or in other words more comprehensive, and the choice of whether to use a purchasing (e.g., churn) model or CLV model depends on the strategy set forth by the company.

7.1 Business Understanding

A home improvement store chain is asking our help with their customer rewards program. In recent years they have been facing steep competition in the market and as a consequence they are experiencing a significant loss in their customer base. Management of the store chain has set forth

a new commercial strategy that is focused on retaining and rewarding their top customers. More specifically, the Chief Marketing Officer (CMO) wants to increase customer loyalty by sending customers a letter to thank them for their business along with a $50 coupon that they can use on their next $100 purchase. Of course this incentive cannot be offered to anybody. If they would send the coupon to all customers they would lose a lot of money, because customers that do not generate much revenue would also come to the store and redeem the coupon. Only the best customers should receive the coupon. In addition the CMO made a strong point in that customers that have spent a lot in the past will not necessarily spend a lot in the future, an observation that is even more true in the home improvement industry. Sending a $50 coupon to customers that will not make any more purchases in the future would, again, mean that the company is losing money unnecessarily. It becomes clear that ranking customers by their past monetary value is not sufficient. Therefore the CMO asks us to predict the future value of their active (purchased in the last 90 days) customers (i.e., compute Customers' Lifetime Value, CLV). Only customers with high CLV should receive the coupon, because it would be very costly if those customers would go to the competition.

7.2 Data Understanding

`http://ballings.co/hidden/aCRM/data/chapter7/customers.csv`

`http://ballings.co/hidden/aCRM/data/chapter7/products.csv`

`http://ballings.co/hidden/aCRM/data/chapter7/stores.csv`

`http://ballings.co/hidden/aCRM/data/chapter7/transactiondetails.csv`

`http://ballings.co/hidden/aCRM/data/chapter7/transactions.csv`

7.3 Data Preparation

7.4 Modeling

7.5 Model Evaluation

7.6 Model Deployment

Bibliography

Ben-Hur, A., Weston, J., 2010. A User's Guide to Support Vector Machines. Methods in Molecular Biology. Department of Computer Science Colorado State University, pp. 223–239.

Berger, P. D., Nasr, N. I., Dec. 1998. Customer lifetime value: Marketing models and applications. Journal of Interactive Marketing 12 (1), 17–30.
URL http://onlinelibrary.wiley.com/doi/10.1002/(SICI)1520-6653(199824)12:1<17::AID-DIR3>3.0.CO;2-K/abstract

Berk, R. A., 2008. Statistical Learning from a Regression Perspective. Springer Series in Statistics. Springer, New York.

Boser, B. E., Guyon, I. M., Vapnik, V. N., 1992. A training algorithm for optimal margin classifiers. Proceedings of the Fifth Annual ACM Workshop on Computational Learning Theory, 144–52.

Breiman, L., Aug. 1996. Bagging predictors. Machine Learning 24 (2), 123–140.

Breiman, L., 1999. Using adaptive bagging to debias regressions. Technical Report, Department of Statistics, University of California, Berkeley.

Breiman, L., Oct. 2001. Random Forests. Machine Learning 45 (1), 5–32.

Breiman, L., Friedman, J., Stone, C. J., Olshen, R. A., Jan. 1984. Classification and Regression Trees, 1st Edition. Wadsworth Statistics/Probability. Chapman and Hall/CRC, New York, N.Y.

Caruana, R., Niculescu-Mizil, A., 2006. An Empirical Comparison of Supervised Learning Algorithms. In: Proceedings of the 23rd international conference on Machine learning. ACM, New York, Pittsburgh, PA, pp. 161–168.

Cederkvist, H. R., Aastveit, A. H., Naes, T., Sep. 2005. A comparison of methods for testing differences in predictive ability. Journal of Chemometrics 19 (9), 500–509.

Chapman, P., Clinton, J., Kerber, R., Khabaza, T., Reinartz, T., Shearer, C., Wirth, R., 2000. CRISP-DM 1.0 , Step-by-step data mining guide.
URL http://www.crisp-dm.org/.

Coussement, K., Van den Poel, D., Apr. 2008. Integrating the voice of customers through call center emails into a decision support system for chum prediction. Information & Management 45 (3), 164–174, wOS:000255813800003.

Cutler, D. R., Edwards, T. C., Beard, K. H., Cutler, A., Hess, K. T., Nov. 2007. Random forests for classification in ecology. Ecology 88 (11), 2783–2792.

De Bock, K. W., Van den Poel, D., Jun. 2012. Reconciling performance and interpretability in customer churn prediction using ensemble learning based on generalized additive models. Expert Systems with Applications 39 (8), 6816–6826.

Deerwester, S., Dumais, S., Furnas, G., Landauer, T., Harshman, R., Sep. 1990. Indexing by Latent Semantic Analysis. Journal of the American Society for Information Science 41 (6), 391–407, wOS:A1990DX15600001.

Demšar, J., 2006. Statistical comparisons of classifiers over multiple data sets. Journal of Machine Learning Research 7, 1–30.

Dietterich, T. G., 1998. Approximate Statistical Tests for Comparing Supervised Classification Learning Algorithms. Neural Computation 10 (7), 1895–1923.

Dietterich, T. G., 2000. Ensemble methods in machine learning. In: Kittler, J., Roli, F. (Eds.), Multiple Classifier Systems. Vol. 1857. pp. 1–15.

Domingos, P., Oct. 2012. A Few Useful Things to Know About Machine Learning. Communications of the Acm 55 (10), 78–87.

Drummond, C., Holte, R. C., Oct. 2006. Cost curves: An improved method for visualizing classifier performance. Machine Learning 65 (1), 95–130.

Dudoit, S., Fridlyand, J., Speed, T. P., Mar. 2002. Comparison of discrimination methods for the classification of tumors using gene expression data. Journal of the American Statistical Association 97 (457), 77–87.

Dwyer, R. F., 1997. Customer lifetime valuation to support marketing decision making. Journal of Interactive Marketing 11 (4), 6–13.
URL http://www.sciencedirect.com/science/article/pii/S1094996897707539

Fan, G., 2009. Kernel-Induced Classification Tree and Random Forest. Technical Report, Dept. of Statistics and Actuarial Science, University of Waterloo.

Fernandez-Delgado, M., Cernadas, E., Barro, S., Amorim, D., Oct. 2014. Do we Need Hundreds of Classifiers to Solve Real World Classification Problems? Journal of Machine Learning Research 15, 3133–3181.

Flach, P., Hernandez-Orallo, J., Ferri, C., 2011. A Coherent Interpretation of AUC as a Measure of Aggregated Classification Performance. In: Proceedings of the 28th International Conference on Machine Learning. Bellevue,WA, USA, p. 8.

Freund, Y., Sep. 1995. Boosting a Weak Learning Algorithm by Majority. Information and Computation 121 (2), 256–285.

Freund, Y., Schapire, R., 1996. Experiments with a new boosting algorithm. In: Machine Learning. Proceedings of the Thirteenth International Conference (ICML '96). Bari, Italy, pp. 148–156.

Friedman, J., Hastie, T., Tibshirani, R., Apr. 2000. Additive logistic regression: A statistical view of boosting. Annals of Statistics 28 (2), 337–374, wOS:000089669700001.

Friedman, J. H., Oct. 2001. Greedy function approximation: A gradient boosting machine. Annals of Statistics 29 (5), 1189–1232.

Friedman, J. H., Feb. 2002. Stochastic gradient boosting. Computational Statistics & Data Analysis 38 (4), 367–378.

Friedman, J. H., Meulman, J. J., May 2003. Multiple additive regression trees with application in epidemiology. Statistics in Medicine 22 (9), 1365–1381.

Fumera, G., Roli, F., Jun. 2005. A theoretical and experimental analysis of linear combiners for multiple classifier systems. Ieee Transactions on Pattern Analysis and Machine Intelligence 27 (6), 942–956.

Gareth, J., Witten, D., Hastie, T., Tibshirani, R., 2013. An Introduction to Statistical Learning with application in R. Springer Texts in Statistics. Springer.

Hand, D. J., Sep. 2005. Good practice in retail credit scorecard assessment. Journal of the Operational Research Society 56 (9), 1109–1117.

Hand, D. J., Anagnostopoulos, C., Apr. 2013. When is the area under the receiver operating characteristic curve an appropriate measure of classifier performance? Pattern Recognition Letters 34 (5), 492–495, wOS:000316425800007.

Hastie, T., Tibshirani, R., Friedman, J., 2009. The Elements of Statistical Learning - Data Mining, Inference, and Prediction, second edition Edition. Springer Series in Statistics. Springer.

Hilden, J., Gerds, T. A., 2013. A note on the evaluation of novel biomarkers: do not rely on integrated discrimination improvement and net reclassification index. Statistics in Medicine, n/a–n/a.

Ishwaran, H., Blackstone, E. H., Pothier, C. E., Lauer, M. S., Sep. 2004. Relative risk forests for exercise heart rate recovery as a predictor of mortality. Journal of the American Statistical Association 99 (467), 591–600.

Jäkel, F., Schölkopf, B., Wichmann, F. A., Dec. 2007. A tutorial on kernel methods for categorization. Journal of Mathematical Psychology 51 (6), 343–358.

Larivière, B., Van den Poel, D., 2005. Predicting customer retention and profitability by using random forests and regression forests techniques. Expert Systems with Applications 29 (2), 472–484.

Liaw, A., 2014. personal communication.

Liaw, A., Wiener, M., 2012. R package randomForest: Breiman and Cutler's random forests for classification and regression.

Luo, T., Kramer, K., Goldgof, D., Hall, L., Samson, S., Remsen, A., Hopkins, T., 2004. Recognizing Plankton Images from the Shadow Image Particle Profiling Evaluation Recorder. IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics 34 (4), 1753 –1762.

Masand, B., Datta, P., Mani, D. R., Li, B., Jun. 1999. CHAMP: A prototype for automated cellular churn prediction. Data Mining and Knowledge Discovery 3 (2), 219–225.

Muenchen, R. A., Aug. 2011. R for SAS and SPSS Users. Springer Science & Business Media.

Murphy, A. H., Winkler, R. L., 1977. Reliability of Subjective Probability Forecasts of Precipitation and Temperature. Journal of the Royal Statistical Society 26 (1), 41–47.

Neslin, S. A., Gupta, S., Kamakura, W., Junxiang Lu, Mason, C. H., May 2006. Defection Detection: Measuring and Understanding the Predictive Accuracy of Customer Churn Models. Journal of Marketing Research (JMR) 43 (2), 204–211.

Noble, W. S., Dec. 2006. What is a support vector machine? Nature Biotechnology 24 (12), 1565–1567.

Park, J. I., Liu, L., Ye, X. P., Jeong, M. K., Jeong, Y.-S., Jan. 2012. Improved prediction of biomass composition for switchgrass using reproducing kernel methods with wavelet compressed FT-NIR spectra. Expert Systems with Applications 39 (1), 1555–1564.

Qi, J., Zhang, L., Liu, Y., Li, L., Zhou, Y., Shen, Y., Liang, L., Li, H., Apr. 2009. ADTreesLogit model for customer churn prediction. Annals of Operations Research 168 (1), 247–265.

R Core Team, R., 2015a. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria.
URL `https://www.R-project.org/`

R Core Team, R., 2015b. R Language Definition: Environments.
URL `https://cran.r-project.org/doc/manuals/r-devel/R-lang.html#Environment-objects`

R Core Team, R., 2015c. R Language Definition: Function objects.
URL `https://cran.r-project.org/doc/manuals/r-devel/R-lang.html#Function-objects`

R Core Team, R., 2015d. R Language Definition: Lexical environment.
URL `https://cran.r-project.org/doc/manuals/r-devel/R-lang.html#Lexical-environment`

Rodriguez, J. J., Kuncheva, L. I., Oct. 2006. Rotation forest: A new classifier ensemble method. Ieee Transactions on Pattern Analysis and Machine Intelligence 28 (10), 1619–1630, wOS:000239605500006.

Schapire, R., Jun. 1990. The Strength of Weak Learnability. Machine Learning 5 (2), 197–227, wOS:A1990DR72200005.

Schölkopf, B., Smola, A., 2002. Learning with Kernels, Support Vector Machines, Regularization, Optimization, and Beyond. MIT Press, Cambridge, MA.

Shawe-Taylor, J., Cristianini, N., 2004. Kernel Methods for Pattern Analysis. Cambridge University Press, Cambridge, UK.

Tibshirani, R., 1996. Regression shrinkage and selection via the Lasso. Journal of the Royal Statistical Society Series B-Methodological 58 (1), 267–288, wOS:A1996TU31400017.

Vert, J.-P., Tsuda, K., Schölkopf, B., 2004. A primer on Kernel Methods. In: Schölkopf, B., Tsuda, K., Vert, J.-P. (Eds.), Kernel Methods in Computational Biology. Computational Molecular Biology. MIT Press, pp. 35–70.

Wilcoxon, F., 1945. Individual comparisons by ranking methods. Biometrics 1, 80–83.

Wolpert, D. H., Oct. 1996. The lack of A priori distinctions between learning algorithms. Neural Computation 8 (7), 1341–1390.

Wolpert, D. H., Macready, W. G., Apr. 1997. No free lunch theorems for optimization. IEEE Transactions on Evolutionary Computation 1 (1), 67–82.

Zadrozny, B., Elkan, C., 2002. Transforming Classifier Scores into Accurate Multiclass Probability Estimates. In: Proceedings of the 8th International Conference on Knowledge Discovery and Data Mining. ACM Press, Edmonton, pp. 694–699.

Zhou, Z.-H., 2012. Ensemble Methods: Foundations and Algorithms. Machine Learning & Pattern Recognition Series. Chapman & Hall/CRC, Boca Raton FL.

About the authors

Michel Ballings (PhD) is Assistant Professor of Business Analytics at The University of Tennessee (Knoxville). He teaches Data Mining and Customer Analytics. His research interests are in the field of social media analytics, customer analytics, machine learning, data mining, and analytical CRM (Customer Relationship Management). He has co-authored several peer- reviewed publications in journals such as European Journal of Operational Research, Omega, Decision Support Systems, and Expert Systems with Applications. He is co-author of several *R* packages, such as `dummy`, `imputeMissings`, `AggregateR`, `hybridEnsemble`, `kernelFactory`, `rotationForest`, `interpretR`, `AUC`, and `lift`.

Dirk Van den Poel (PhD) is a Professor of Data Analytics/Big Data at Ghent University, Belgium. He teaches courses such as Statistical Computing, Big Data, Analytical Customer Relationship Management, Advanced Predictive Analytics, Predictive and Prescriptive Analytics. He co-founded the advanced Master of Science in Marketing Analysis, the first (predictive) analytics master program in the world as well as the Master of Science in Statistical Data Analysis and the Master of Science in Business Engineering/-Data Analytics. His major research interests are in the field of analytical CRM: customer acquisition, churn, upsell/cross-sell, and win-back modeling. His methodological interests include ensemble classification methods and big data analytics.

He has co-authored 80+ peer-reviewed (ISI-indexed) publications in journals such as Journal of Statistical Software, IEEE Transactions on Power Systems, Journal of Applied Econometrics, Applied Geography, and European Journal of Operational Research.