Τμήμα Μηχανικών
Πληροφορικής ΑΤΕΙΘ

DATA PREPARATION AND PREPROCESSING FOR DATA MINING USING R

by

Pazaras Christos

A thesis submitted in partial fulfillment of the requirements for the degree of

INFORMATION TECHNOLOGY ENGINEER

at the

Alexander Technological Educational Institute of Thessaloniki

Supervisor: Dimitrios A. Dervos

December 2013

Table of Contents

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisor, Professor Dimitrios A. Dervos, for his continuous support of my thesis study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance was crucial to me during the times of research and writing of this thesis.

Επιτομή

Η παρούσα πτυχιακή εργασία εκπονήθηκε ως αποτέλεσμα μελέτης και έρευνας που πραγματοποιήθηκε στα πλαίσια της διερεύνησης της γλώσσας προγραμματισμού «R» και ειδικότερα των δυνατοτήτων αυτής στους τομείς της εξόρυξης πληροφορίας και της συνδεσιμότητάς της με Συστήματα Διαχείρισης Βάσεων Δεδομένων. Η γλώσσα προγραμματισμού «R» μελετήθηκε σε βάθος και η μελέτη επικεντρώθηκε αρχικά στα θεμελιώδη στοιχεία του προγραμματισμού με αυτή. Μετέπειτα μελετήθηκαν οι δυνατότητές της σε ζητήματα εξόρυξης πληροφορίας, σε συνδυασμό με τις δυνατότητες συνδεσιμότητάς της με δημοφιλή Συστήματα Διαχείρισης Βάσεων Δεδομένων και ανταλλαγής πληροφοριών με αυτά. Στην προσπάθεια να ενισχυθεί η κατανόηση των σύνθετων θεμάτων που διερευνήθηκαν πραγματοποιήθηκαν δύο περιπτωσιολογικές μελέτες: Η πρώτη επικεντρώθηκε στην εξόρυξη κανόνων συσχέτισης από ένα σύνολο δεδομένων το οποίο περιέχει αρχεία παρελθόντων συναλλαγών ενός πολυκαταστήματος, οι οποίοι χρησιμοποιήθηκαν στη συνέχεια ως βάση για την κατασκευή ενός συστήματος συστάσεων – ενός συστήματος το οποίο έχει την ικανότητα να προτείνει στοχευμένα προϊόντα στους πελάτες του πολυκαταστήματος βασιζόμενο στα αρχεία συναλλαγών του παρελθόντος του πολυκαταστήματος. Η μελέτη αυτή είναι μια προσομοίωση του tutorial της IBM “Mining your business in retail with IBM DB2 Intelligent Miner”[22] και τα αποτελέσματά της αντικατοπτρίζουν πλήρως τα αποτελέσματα που παράγονται στο tutorial αυτό, επαληθεύοντας έτσι την ικανότητα της R να πραγματοποιεί επιτυχώς και με ακρίβεια εξόρυξη κανόνων συσχέτισης και συνδεσιμότητα με Συστήματα Διαχείρισης Βάσεων Δεδομένων. Η δεύτερη μελέτη εστίασε στην κατασκευή ενός δέντρου αποφάσεων με σκοπό την κατηγοριοποίηση του συνόλου δεδομένων των επιβατών του Τιτανικού [13] χρησιμοποιώντας τον αλγόριθμο C4.5. Και οι δύο μελέτες αποδεικνύουν ικανοποιητικά ότι η R είναι μία ισχυρή γλώσσα προγραμματισμού, ικανή να συνδεθεί και να ανταλλάξει πληροφορίες με Συστήματα Διαχείρισης Βάσεων Δεδομένων και να εκτελέσει σύνθετες εργασίες εξόρυξης πληροφορίας.

Abstract

This thesis was conducted as a result of study and research done for the sakes of the exploration of the R programming language and especially its data mining and DBMS connectivity capabilities. The R programming language was studied in detail, the study initially focusing on the fundamental elements of programming with it. Furthermore, R's data mining capabilities were examined, along with its capability of connecting to and exchanging information with popular DBMSs. To aid the understanding of the complex issues investigated, verify the research results and extend the experience gained, two case studies were conducted: The first one was focused on mining association rules from a supermarket transaction records dataset using R, with which a product recommender system was created – a system that can accurately recommend products to the customers according to the previous transactions records of the supermarket. This case study is a simulation of IBM's tutorial "Mining your business in retail with IBM DB2 Intelligent Miner"[22] and the results produced by R totally reflect the results produced in the tutorial, thus verifying R's capability of successfully and accurately performing association rules mining and DBMS connectivity. The second case study was focused on creating a decision tree to classify the passengers of the Titanic dataset [13] using the C4.5 algorithm. Both case studies sufficiently prove that R is a strong programming language, capable of performing DBMS connectivity and complex data mining tasks.

Introduction

The following thesis was conducted in order to explore the capabilities of the R language with regards to data mining and to connecting to, and exchanging information with, popular DBMSs such as MySQL and PostgreSQL.

The first chapter covers everything a new R user needs to know about R – its environment, its commands syntax, its data types, its behavior. Further on, it focuses on certain important aspects and characteristics of the language as well as several key features that it offers, such as the function vectorization ability. Most of the information described on the first chapter arises from “A (not so) short introduction to S4: Object Programming in R” by Genolini, C. [10] and “Data Mining With R: Learning with case studies” by Torgo, L. [12].

Moving on, the second chapter introduces the term “data mining” and describes its purpose, its uses, what it offers, how it is performed. It then focuses on two major data mining techniques, the association rule mining technique and the classification technique, which are further analyzed. Finally, ways in which these data mining methods can be performed in the R language environment are described and demonstrated. The information described on this chapter was based on “Association Rule Mining: A Survey” by Zhao, Q et al[29] and “R and Data Mining: Examples and Case Studies” by Zhao, Y. [30].

The third chapter is devoted to introducing the term “DBMS” and describing their usability, the features they provide in terms of data integrity and availability, and how manually fetching information from a database differs from producing information using data mining methods, how a DBMS cannot provide the information that can be gained via data mining. At that point, it focuses on R’s connectivity ability with two popular DBMSs, MySQL and PostgreSQL, and describes the ways that R can connect to those DBMSs and exchange information with them, thus filling the gap between the DBMSs and their data mining capabilities.

The fourth chapter is focused on the setting up and configuration of a virtual machine on which the case studies for the sakes of this thesis were conducted. Detailed information is provided in a step-by-step manner upon installing and configuring everything that is needed in order for the machine to be able to be used for data mining using R and MySQL or PostgreSQL.

Finally, the fifth chapter is devoted on the presentation of two case studies that were conducted, in the first of which the procedure of mining association rules from a supermarket transaction records dataset using R is demonstrated, while in the second of which the procedure of the classification of a dataset containing information about the passengers of the Titanic according to their surviving the

tragedy or not using R is demonstrated. Both case studies were conducted on the machine that was set up on the fourth chapter.

1. R

1.1 Summary

R is a free, open-source programming language used for statistical computing, graphics, and data analysis. It is widely used by statisticians and data scientists. Polls and surveys of data miners show that R's popularity has increased substantially in recent years [23].

R is an implementation of the S programming language, which was created by John Chambers and colleagues during his time at Bell Labs (formerly AT&T, now Lucent Technologies), combined with lexical scoping semantics inspired by Scheme. It can be considered a different implementation of S - there are some important differences, but much of the code written for S runs unaltered under R. It was created at the University of Auckland, New Zealand, by Robert Gentleman and Ross Ihaka, and it is currently being developed by the R Development Core Team, of which Chambers is a member. As is easily perceived, the name of the language came partly from the creators' last names initial letter, and partly as a play on the name of S.

The source code for the R software environment is written primarily in C, Fortran, and R. Though R itself is an interpreted language and as such uses a command line interface, several graphical user interfaces have been developed and are available for use with R.

R is available as Free Software under the terms of the Free Software Foundation's GNU General Public License in source code form. It compiles and runs on a wide variety of UNIX platforms and similar systems (including FreeBSD and Linux), Windows and MacOS [10].

1.2 Why use R

A fair question one might ask is what makes R worth learning and using. The following bullet points will try to provide the answer:

- R is free software, distributed under the Free Software Foundation's GNU General Public License.
- R is a powerful data-analysis package with many standard and cutting-edge statistical functions. (http://en.wikipedia.org, 2013)
- R is easily extensible, via user-defined functions and packages.
- Most of R's standard functions are written in R itself, which makes it easy for users to follow the algorithmic choices made.
- C, C++, and Fortran code can be called and executed during run time.
- R objects can be directly manipulated by C, C++ or Java.

- R is available for all major operating systems (Windows, Mac OS, GNU-Linux).
- R is object-oriented.
- A portable version of R can be installed on a USB stick.

1.3 Uses and popular applications

R provides a wide variety of statistical and graphical techniques, which include time series analysis, classical statistical tests, classification, clustering, linear and nonlinear modeling, and is highly extensible. It shines due to the ease of well-designed publication-quality plots generation that it provides, including mathematical symbols and formulae where needed. It can be used for graphical statistical charts generation, data mining tasks on data, complex mathematical algorithms, statistical analyses, probability measuring, and much more.

R is widely used in political science, statistics, econometrics, actuarial sciences, sociology, finance, and elsewhere [23].

1.4 The R environment

Many users think of R as a statistics system. We prefer to think of it of an environment within which statistical techniques are implemented [2].

R is a complete suite of software facilities capable of manipulating data, calculating results and graphically displaying them. These facilities include:

- a collection of graphical data displaying facilities
- a collection of data analysis tools
- facilities for data handling and storing
- a simple and effective programming language, which includes all of the most common control structures a typical programming language implements (conditionals, loops, input/output operations, user-defined functions, recursion)
- operators for performing calculations on arrays / matrices

Using the term "environment", R is intended to be characterized not as a mere accretion of inflexible, highly specific tools (as is the case with most data analysis software), but as a totally coherent system. It is designed around a true computer programming language, allowing users to define their own functions and packages, execute code from other programming languages such as C or Fortran, and thus contributing to the extension of the language's functionality.

R's extensionality is hugely supported and amplified by the Comprehensive R Archive Network (CRAN), which hosts an R package repository containing

packages available to users for downloading and installation. Currently, the CRAN package repository features 4986 available packages (http://cran.r-project.org, October 2013). The packages hosted by CRAN are tested regularly, and users can contribute their own packages to the repository, thus enhancing the functionality of R.

1.4.1 Commands syntax

R is an object oriented programming language. This means that virtually everything can be stored as an R object. Each object has a class. This class describes what the object contains and what each function does with it. For instance, plot(x) produces different outputs depending on whether x is a regression object or a vector [14].

R programming - syntax principles:

- The assignment symbol is "<-". Alternatively, the classical "=" symbol can be used, but use of "<-" is encouraged.
- Arguments are passed to functions inside parentheses.
- R supports function combination. For instance, it is legal to type:

```
> mean(rnorm(1000)^2)
```

- Line comments use the "#" (hash / pound) symbol. For example:

```
> # this is a comment
> 1+1   # this is also a comment
```

- Commands are normally seperated by a newline, except for a multiple statements on one line case, where commands are seperated using the";" (semicolon) symbol. One statement on multiple lines is also legal. In this case, R uses the "+" (plus) character to specify the lines containing one statement. For example:

```
> x <- 3+5; y <- 8+6; z <- 10
> x <-
+ 3+5; y
+ <-
+ 8+6; z <- 10
```

- R is a case-sensitive language
- The "_" (underscore) symbol is generally not used in names, but rather the "." (dot) character is used. The reasoning behind this is traditional though, and not language restriction originating in any way - it is perfectly legal to use underscores in names.

1.4.2 Issuing a command

R commands are entered in the R console. The commands are typed after the angle bracket symbol ">", and are issued by pressing the "Enter" key on the keyboard. When a command is issued, R responds with the resulting output, printed right under the line where the command was typed. For example, if a negation command is issued, R will respond with the result of the negation:

```
> 5 - 2
[1] 3
```

If the output is not in text form (for example, if a plotting command is issued), then the appropriate result will appear (in the example case of a plotting command, an appropriate graph will appear as the result).

1.4.3 R functions

As previously mentioned, one of the greatest strengths of R is its ability of allowing the users to extend its functionality by adding their own, user-defined functions.

Functions in R are treated as objects, which have the mode “function”. This means that the R interpreter is able to pass control to the function, along with arguments that may be necessary for the function to accomplish the actions that are desired. The function in turn must correctly perform its task and return control to the interpreter as well as any results which may be stored in other objects [8].

To define a function, its source code must be assigned to a word, which will be the name of the function. Similarly to most programming languages, the function’s parameters are enclosed in parentheses, and its source code is enclosed between “{“ and “}” brackets. For example:

```
> hello <- function(some.parameter=some.value) {
+     3+3
+ }
```

When the function “hello” is called, the result seen on the console will be:

```
[1] 6
```

R functions facts:

- R supports calling a function inside another function.
- R allows the user to define a function inside another function.
- R supports both matching a function’s parameters by name and matching them by position, and also a mix of the two matching modes. For instance:

```
> hello <- function(x,y,z) {
+     # do something
+ }
```

Function “hello(x,y,z)” can be called either by positional matching:

```
> hello(1,2,3)
```

which will result in x being assigned a value of 1, y being assigned a value of 2, and z being assigned a value of 3, or by the parameters’ name matching:

```
> hello(y=2,z=3,x=1)
```

Finally, function “hello(x,y,z)” can be called by mixing the two modes:

```
> hello(1,z=3,y=2)
```

which will, again, result in x being assigned a value of 1, y being assigned a value of 2, and z being assigned a value of 3. When using mixed matching modes, each parameter that is matched by name is removed from the parameters list, and all the remaining unnamed arguments are matched in the order that they are listed in the function definition.

1.4.4 Vectorization

A big advantage of R over other statistical analysis software worth mentioning is the ability it gives the user to simply program a series of analyses which will then be executed successively, instead of having to navigate through confusing drop-down menus and dialog boxes. In most major programming languages, when the programmer needs to perform a series of operations on a certain set of data, the most common approach will be the use of a loop structure, via which the programmer will navigate through all of the data, and for each part of it perform the needed operations.

In R, this procedure can be really simplified – or even completely omitted - due to a very useful feature R offers, called “Vectorization”. Vectorization, in short, refers to the operation of a function not only applying to a single value, but to a whole set of data. Vectorized functions work not only on a single value, but on all the values of, for example, a vector, thus making the use of a loop to perform the function operation on each of the vector’s components one by one quite unnecessary.

Apart from easing the programming, vectorization greatly optimizes the code speed as well.

The use of the vectorization feature is demonstrated in the following examples:

```
> x <- c(1,2,3,4,5,6,7,8,9)
> sum(x)

[1] 45
```

In another programming language, to find the sum of all the vector's elements the programmer would probably have to use a loop and a separate variable where the sum would be stored, and in each loop increase the sum variable by the value of the component being accessed in that particular loop. With vectorization, this procedure is quite unnecessary, since the vectorized function "sum()" performs it.

So, the "vectorized function" term is introduced. A vectorized function is a function that implements vectorization in its operation – it operates on a set of data, a vector for example, separately performing its operations on each of the data - or the vector's components.

Consequently, if a vectorized function that performs the task the programmer needs exists, the programmer is freed of the labor to execute the function for each component of the data set in use.

An important relevant issue here is what happens when a vectorized function which will perform the task the programmer needs does not exist. Is the looping-and-applying method described earlier the only way to go?

The answer is, luckily, no. R has predicted this, and has a whole family of functions devoted to this matter, known as the "apply family" functions.

The apply functions are a set of functions that will execute any given function on each element of a set of data. Thus, they can turn any function to a vectorized one. Though there exist many apply functions, each having its own purpose, deep down what they all provide is a vectorization of other functions.

The following example demonstrates the vectorization of the "class()" function, which prints the class of the given object (for example: numeric, factor, data.frame, etc), using the "lapply()" function.

```
> x <- matrix(data=1:9,nrow=3)
> x

     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9

> class(x)

[1] "matrix"

> class(x[1,1])
```

```
[1] "integer"

> lapply(x,class)

[[1]]
[1] "integer"

[[2]]
[1] "integer"

[[3]]
[1] "integer"

[[4]]
[1] "integer"

[[5]]
[1] "integer"

[[6]]
[1] "integer"

[[7]]
[1] "integer"


[[8]]
[1] "integer"

[[9]]
[1] "integer"
```

Thus, with the use of the “lapply()” function, the class function was executed 9 times, one for each item of the “x” matrix.

Apart from the “lapply()” function, more variations of the apply function exist. The most basic of them are:

- apply() – used for matrices, returns a matrix. The “MARGIN” parameter needs to be set, which defines whether the function should consider the rows(c(1)), the columns(c(2)), or both (c(1,2)).
- lapply() – returns a list
- sapply() – similar to lapply(), returns a vector rather than a list
- mapply() – used to apply a function to a set of several data structures. The function is first applied to the first elements of each structure, then to the second elements, then the third, etc. Returns a character (what is known in other programming languages as a string)
- tapply() – Used to apply a function to each element of a nested list structure, recursively. Returns a vector of integers.

1.4.5 R scoping

Since the time of creation, R has always been lexically scoped. Objects and functions in R use lexical scoping.

1.4.6 Reading commands from external source file

R can read and execute source code from external sources. Quite a useful feature, since the source code can be kept organized in separate R scripts, and thus the need of certain commands to be reentered every time the R command line is eliminated as well. To read an external R script, the “source” function needs to be called, as such:

```
> source(“filename.R”)
```

When the “source” function is used, R will attempt to read and interpret the code from the file supplied. If an error occurs, the script reading procedure will stop, and R will print an error message on the console describing the error. Otherwise, the angle bracket symbol “>” will appear again, which will mean that the script was successfully read and interpreted.

1.4.7 Importing and exporting data

Generally speaking, data can be stored in a large variety of formats. Almost every statistical analysis software product utilizes its own format. R can read almost all file formats, thanks to a variety of functions it includes.

Although more on importing and exporting data will be covered on a subsequent chapter, a reference to the functions used to read data from some of the most common formats is worth being made.

One of the most common formats of storing tabular data (i.e. numbers and text) is the well-known “csv” (Comma Separated Values, or Character Separated Values) format. Csv files store data in plain text form and, as the format name specifies, generally use the “,” (comma) character to separate the values. Typically, each row of a csv file is in most cases considered a unique entity, a record, and each column is considered a “field”, though these admissions are not obligatory.

To read data from a csv file, R provides the “read.csv()” function. The csv file in question has to be specified in the parameter list, and other parameters can optionally be specified as well, like the “HEADER” parameter, which, if set to TRUE, considers the first line of the csv file to be a line containing the column names, not a line containing data, or the “colClasses” parameter, which specifies

the desired class that each column will be read as. For example, if the csv file to be read consists of two columns which contain numeric values and it is desired that they are read in R as characters, then, instead of reading them as numeric and converting them to characters afterwards, the “read.csv()” function can be instructed to read them directly as characters – i.e. perform the numeric to character conversion during read time, “on the fly” – by making use of the “colClasses” parameter. In this case, the function call would be:

```
> read.csv(“<csv.file>”,
+ colClasses=c(“character”,”character”))
```

Apart from the “csv” format, tabular data can also be read from any file as long as the structure of the data inside that file is instructed to R. The function used in such cases is the “read.table()” function. The “read.table()” function accepts a long list of parameters, most of which have to be defined in order for the data to be read correctly.

Data in R can be read in various other ways and from a variety of other sources. Such a source could be a network connection, for example, over which data can be read via making use of the “read.socket()” function. Importing and exporting data to and from R will be covered in more detail later on.

1.4.8 Workspace and memory management

R provides a fair amount of control over the work environment, the workspace objects and the memory usage:

- Workspaces can be saved as workspace images (for example, at the end of an R session) and reloaded when necessary.
- Workspace objects can be saved directly to the hard drive instead of the memory.
- History can be saved to external files and reloaded when needed.
- Memory usage limits can be enforced, so that R does not use more memory than the user intends to.

Due to all of the facts above, and with the right workspace and memory management and handling, R can effectively process big data sets and execute highly memory demanding functions even in weak, not so powerful systems.

1.5 R data types

To do anything in R, one has to learn the data types that R can handle. While common data types found in most major programming languages - such as integers, doubles, strings, logicals – do exist in R as well, some additional types of data which R utilizes need to be defined and explained.

1.5.1 Vectors

Vectors are the simplest R objects, an ordered list of primitive R objects of a given type (e.g. real numbers, strings, logicals). Vectors are indexed by integers starting at 1. What is important to note is that a vector consists of a sequence of data elements of the same basic type. The members of a vector are officially called the vector's *components*.

To create a vector, the "c()" function must be used, as follows:

```
> c(1,2,3,4)
```

```
[1] 1 2 3 4
```

A vector can also be created using the ":" symbol, which defines a number range, or the "seq()" function, which also defines a number range and allows the specification of an interval between the numbers using the "by" parameter. The above functions are used as such:

```
> c(1:5)
```

```
[1] 1 2 3 4 5
```

```
> seq(1,5,by=2)          # the "seq()" function returns a vector
                         # by itself
```

```
[1] 1 3 5
```

Vectors can be assigned in variables:

```
> x <- c(1,2,3,4)
```

To refer to a component of a vector, its index must be used, as such:

```
> x <- c(1,2,3,4)
> x[1]
```

```
[1] 1
```

1.5.2 Factors

Factors are a similar data structure to vectors, except for the fact that a factor holds categorical elements. Each element of a factor belongs to a “level” of that factor. To create a factor, the “factor()” function should be used, which converts a vector into a factor. For instance:

```
> factor(c(“windy”,“windy”,“windy”,”sunny”,“windy”,”sunny”))

[1] windy windy windy sunny windy sunny
Levels: sunny windy
```

Thus a factor has been created, consisting of 2 levels, “sunny” and “windy”. Factors are really useful for storing categorical data, due to their “levels” mechanism.

The levels of a factor can be obtained using the “levels()” function, as follows:

```
> x <-
+ factor(c(“windy”,“windy”,“windy”,”sunny”,“windy”,”sunny”))
> levels(x)

[1] "sunny" "windy"
```

The number of instances of each level in a factor can be counted via the “table()” function:

```
> x <-
+ factor(c(“windy”,“windy”,“windy”,”sunny”,“windy”,”sunny”))
> table(x)

sunny windy
    2     4
```

Levels can have custom names or labels, via the “levels” and “labels” parameters. The “levels” parameter is used to manually specify the levels of the factor, whereas the “labels” parameters is used to give a label to each level. For examples:

```
> factor(c(“rainy”,“windy”,“windy”,”sunny”,“rainy”,”sunny”),
levels=c(”rainy”,”sunny”))
> x

[1] rainy <NA>  <NA>  sunny rainy sunny
Levels: rainy sunny

> table(x)

X
rainy sunny
    2     2
```

```
> x <-
+ factor(c("rainy","windy","windy","sunny","rainy","sunny"),
+ labels=c("rainy","sunny","windy"))
> x

[1] rainy windy windy sunny rainy sunny
Levels: rainy sunny windy

> table(x)

x
rainy sunny windy
    2     2     2
```

As is easily perceived, in the case of manually specifying the levels, all instances of variables that do not belong to a level specified are ignored.

1.5.3 Matrices

A matrix is a similar to a vector, but with a specific layout format, such that it looks like an actual matrix - i.e. the elements of a matrix are indexed by two integers, each starting at 1.

Matrix elements can only be of numeric or character type. To create a matrix, one way is to use the “matrix()” function. A vector of data has to be entered, along with the numbers of rows and columns, and optionally the way that R will read the matrix can be specified as well, which can be either by row or by column (the default mode). The read mode is specified by the logical parameter “byrow”. Below is an example of creating a matrix:

```
> matrix(data=1:16,nrow=4,ncol=4,byrow=TRUE)

     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
[4,]   13   14   15   16
```

A matrix can be created by combining multiple vectors, in case the data about to be entered in a matrix is stored in vectors. To create a matrix by combining multiple matrices, the functions “rbind()” and “cbind()” can be used. Function “rbind()” will combine the vectors in a “row-by-row” mode, whereas function “cbind()” will combine the vectors in a “column-by-column” mode. For example:

```
> v1 <- c(1,2,3,4,5)
> v2 <- c(6,7,8,9,10)
> v3 <- c(11,12,13,14,15)
> rbind(v1,v2,v3)

   [,1] [,2] [,3] [,4] [,5]
v1    1    2    3    4    5
```

```
v2    6    7    8    9   10
v3   11   12   13   14   15

> cbind(v1,v2,v3)

     v1 v2 v3
[1,]  1  6 11
[2,]  2  7 12
[3,]  3  8 13
[4,]  4  9 14
[5,]  5 10 15
```

The dimension of a matrix can be obtained using the "dim()" function. Alternatively, "nrow()" and "ncol()" functions return the number of the rows and columns, respectively, of a matrix. For example:

```
> mx <- matrix(data=1:16,nrow=4,ncol=4,byrow=TRUE)
> dim(mx)

[1] 4 4

> nrow(mx)

[1] 4

> ncol(mx)

[1] 4
```

Using the "t()" function, a matrix can be transposed, i.e. its columns will become its rows, and its rows will become its columns. Function "t()" is used as follows:

```
> mx <- matrix(data=1:16,nrow=4,ncol=4,byrow=TRUE)
> mx

     [,1] [,2] [,3] [,4]
[1,]    1    2    3    4
[2,]    5    6    7    8
[3,]    9   10   11   12
[4,]   13   14   15   16

> t(mx)

     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16
```

The rows and columns of a matrix can be named, using the functions "rownames ()" and "colnames ()", respectively:

```
> mx <- matrix(data=1:16,nrow=4,ncol=4,byrow=TRUE)
> rownames(mx) <- c("row1","row2","row3","row4")
> mx
```

```
     [,1] [,2] [,3] [,4]
row1    1    2    3    4
row2    5    6    7    8
row3    9   10   11   12
row4   13   14   15   16

> colnames(mx) <- c("col1","col2","col3","col4")
> mx

     col1 col2 col3 col4
row1    1    2    3    4
row2    5    6    7    8
row3    9   10   11   12
row4   13   14   15   16
```

Finally, reference to a matrix element can be achieved using the "[" and "]" (square brackets) symbols, as in most major programming languages. If the row or the column index is not specified, R will return the same row/column as the row/column specified. If the row or the column index is not specified, but the row and column indexes are separated by a comma (","), then the whole row or column, respectively, will be returned. Usage is performed as follows:

```
> mx <- matrix(data=1:16,nrow=4,ncol=4,byrow=TRUE)
> mx[1]     # Only one index is specified, so R returns the same
row/column as the row/column specified:

[1] 1

> mx[2]     # Only one index is specified, so R returns the same
row/column as the row/column specified:

[1] 5

> mx[1,2]

[1] 2

> mx[,2]    # Row index is not specified, so R returns the whole row:

[1]  2  6 10 14

> mx[2,]    # Column index is not specified, so R returns the whole
column:

[1] 5 6 7 8
```

1.5.4 Arrays

Arrays are extensions of matrices to more than two dimensions. So arrays are similar to matrices, differing in the fact that they can have more than 2 dimensions. An array in R is composed of n dimensions, where each dimension is a vector of R objects of the same type. Items are placed into the array column-wise, and not row-wise as in most major programming languages. So the first item will be placed in position [1,1], the second will be placed in position [2,1], etcetera.

An array of a single dimension can be constructed thus:

```
> arr <- array(c(“a”),dim=c(1))
> arr

[1] "a"
```

Similarly, an array of two dimensions can be constructed thus:

```
> arr <- array(c(“a”,”b”,”c”,”d”),dim=c(2,2))
> arr

     [,1] [,2]
[1,] "a"  "c"
[2,] "b"  "d"
```

If more than two dimensions are required, then a multidimensional array can be constructed as follows:

```
> arr <-
+ array(c("a","b","c","d","e","f","g","h"),dim=c(2,2,2))

> arr

, , 1

     [,1] [,2]
[1,] "a"  "c"
[2,] "b"  "d"

, , 2

     [,1] [,2]
[1,] "e"  "g"
[2,] "f"  "h"
```

The dimensions, row and column numbers, type of the components and components referencing can be achieved the same way it can be achieved in matrices.

1.5.5 Lists

A list is similar to a vector, but the elements need not all be of the same type. It is a collection of R objects of any type. The elements of a list can be indexed either by integers or by named strings, i.e. an R list can be used to implement what is known in other programming languages as an "associative array", "hash table", "map" or "dictionary".

A list can be created via the “list()” function, thus:

```
> x <- 5
> y <- c(TRUE,FALSE,FALSE,TRUE,FALSE)
> z <-
+ array(c("a","b","c","d","e","f","g","h"),dim=c(2,2,2))
> mylist <- list(x,y,z)
> mylist

[[1]]
[1] 5

[[2]]
[1]  TRUE FALSE FALSE  TRUE FALSE

[[3]]
, , 1

     [,1] [,2]
[1,] "a"  "c"
[2,] "b"  "d"

, , 2

     [,1] [,2]
[1,] "e"  "g"
[2,] "f"  "h"
```

The ways of referencing a list's components are quite flexible. Also, all of the methods a list component can be referenced by can be combined. A list component can be referenced by:

- Index number:

```
> x <- 5
> y <- c(TRUE,FALSE,FALSE,TRUE,FALSE)
> mylist <- list(x,y,z)
> mylist[1]

[[1]]
[1] 5
```

- Name string, using the "$" (dollar) sign:

```
> mylist <- list()
> mylist$planet <- "Earth"
> mylist$planet

[1] "Earth"

> mylist["planet"]

$planet
[1] "Earth"
```

- A combination of index numbers and name strings:

```
> x <- 5
> y <- "Earth"
> mylist <- list()
> mylist[1] <- x
```

```
> mylist$planet <- y
> mylist

[[1]]
[1] 5

$planet
[1] "Earth"
```

Direct reference to a list's inner component is possible, too. For example, if a vector is part of a list, a component of this vector can directly be referenced by the list name. For this to be done, the list index number needs to be contained inside double square brackets, thus:

```
> x <- c(1,2,3,4)
> mylist <- list(x)
> mylist[[1]]

[1] 1 2 3 4

> mylist[[1]][1]

[1] 1
```

A list component can be named upon list creation, as follows:

```
> mylist <- list(country=c("Greece","Bulgaria"))
> mylist

$country
[1] "Greece"   "Bulgaria"
```

A list can be transformed into a vector, via the "unlist()" function. Example:

```
> mylist <- list(5,c(1,2,3),matrix(data=1:16,nrow=4,ncol=4))
> mylist

[[1]]
[1] 5

[[2]]
[1] 1 2 3

[[3]]
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16

> mylist <- unlist(mylist)
> mylist
```

1.5.6 Data frames

A data frame is similar to a matrix, but does not assume that all columns contain objects of the same type. It is a list of variables or vectors of the same length. Less formally, a data frame is a type of table where the typical use employs the rows as observations and the columns as variables. Data frames can be really helpful for storing and handling data coming from a database or a dataset.

A data frame is created with the use of the “data.frame()” function, thus:

```
> a <- c(1,2,3)
> b <- c(4,5,6)
> c <- c(7,8,9)
> df <- data.frame(a,b,c)
> df

  a b c
1 1 4 7
2 2 5 8
3 3 6 9
```

The top line of the table, called the header, contains the column names. Each horizontal line afterward denotes a data row, which begins with the name of the row, and is then followed by the actual data. Each data member of a row is called a cell [28].

A data frame’s rows and columns can be named, using the “row.names” parameter for row naming, and either the “names()” function or direct naming for column naming. The column names are stored in the *header* of the data frame. The naming procedure is carried out as follows:

```
> a <- c(1,2,3)
> b <- c(4,5,6)
> c <- c(7,8,9)
> df <-
+ data.frame(c1=a,c2=b,c3=c,row.names=c("row1","row2","row3"))
> df

       c1   c2   c3
row1    1    4    7
row2    2    5    8
row3    3    6    9

> names(df) <- c(“new.col1”,” new.col2”,” new.col3”)
> df

     new.col1 new.col2 new.col3
row1        1        4        7
row2        2        5        8
row3        3        6        9
```

Access to a data frame’s components is obtained in a way similar to the one used to access a list’s components. Either number indexing can be used, where the

numbers are once again contained between square brackets, or name strings can also be used, either contained between square brackets or following the dollar sign. Some examples of accessing a data frame's components:

```
> df <- data.frame(c1=c(1,2,3),c2=c(4,5,6),c3=c(7,8,9))
> df

  c1 c2 c3
1  1  4  7
2  2  5  8
3  3  6  9

> df[1]

  c1
1  1
2  2
3  3

> df[1,1]

[1] 1

> df["c1"]

  c1
1  1
2  2
3  3

> df$c1

[1] 1 2 3
```

1.6 Conclusion

The basic data types, objects and commands needed to create and manipulate data in the R language have been described and their usage has been demonstrated. It is easily perceptible that R is a simple but powerful language. In the next chapter, its usability and performance in performing data mining in sets of data will be described in detail.

2. Data Mining

2.1 Introduction

Nowadays, big data is found everywhere. Corporations keep hiring data scientists, are worried about personal data and data control, entrepreneurs are trying to find new ways to collect data, control it, monetize it. Long story short, data is valuable, data is powerful.

Data mining is inseparably connected to these facts. It is the connection, the link between raw data and intelligence, the means to extract useful information from tons of data. Without data mining, big data would be almost useless – no brain could be able to extract this much intelligence from this much data in such little time.

The term “data mining”, in general, has been given many definitions:

“Generally, data mining (sometimes called data or knowledge discovery) is the process of analyzing data from different perspectives and summarizing it into useful information - information that can be used to increase revenue, cuts costs, or both.” [5]

“Data mining (the analysis step of the "Knowledge Discovery in Databases" process, or KDD), an interdisciplinary subfield of computer science, is the computational process of discovering patterns in large data sets involving methods at the intersection of artificial intelligence, machine learning, statistics, and database systems.” [17]

So data mining refers to a series of actions performed on a set of data, in order to extract intelligence from it. Data is considered to be any number, text or fact that can be processed by a computer, usually stored in a computer database. Information is considered to be any pattern, relationship or association arising from the processing of the data, and the useful subset of these patterns, relationships or associations is used to produce knowledge – information that has a meaning and can be used in a beneficial way.

The term “data mining” was first introduced in the 1990s, but data mining is actually the evolution of a field with a long history.

Data mining roots are traced back along three family lines: classical statistics, artificial intelligence, and machine learning.

Statistics are the foundation of most technologies on which data mining is built, e.g. regression analysis, standard distribution, standard deviation, standard variance, discriminant analysis, cluster analysis, and confidence intervals. All of these are used to study data and data relationships.

Artificial intelligence, or AI, which is built upon heuristics as opposed to statistics, attempts to apply human-thought-like processing to statistical problems. Certain AI concepts which were adopted by some high-end commercial products, such as query optimization modules for Relational Database Management Systems (RDBMS).

Machine learning is the union of statistics and AI. It could be considered an evolution of AI, because it blends AI heuristics with advanced statistical analysis. Machine learning attempts to let computer programs learn about the data they study, such that programs make different decisions based on the qualities of the studied data, using statistics for fundamental concepts, and adding more advanced AI heuristics and algorithms to achieve its goals.

Data mining, in many ways, is fundamentally the adaptation of machine learning techniques to business applications. Data mining is best described as the union of historical and recent developments in statistics, AI, and machine learning. These techniques are then used together to study data and find previously-hidden trends or patterns within [27].

2.2 Data mining phases and techniques

The data mining process is often divided into 5 phases:

- The Selection phase – definition of the dataset that will be processed,
- The Pre-Processing phase – editing of the dataset in order for it to be ready for mining, i.e. have no missing values, be “clean”,
- The Transformation phase – conversion of the data in the appropriate format for it to be entered in the data mining system,
- The Data Mining phase – the data is processed (“mined”) using a data mining algorithm or technique,
- The Evaluation phase – the data mining results are evaluated and knowledge is extracted from them.
  (en.wikipedia.org)

Each of these phases plays its own specific role in the data mining procedure. The Pre-Processing phase is very important, as the result of the data mining process is highly dependent on it. The Data Mining phase is the one where the actual data mining happens, and as such, the phase that will be subsequently analyzed.

The Data Mining phase, as described above, is the phase where the data is processed for information, or mined, making use of a data mining algorithm or technique. Such algorithms and techniques have been researched and developed over the past years, finally having come down to the conclusion that generally, when data mining, four main types of relationships among data are sought:

Associations, classes, clusters, and sequential patterns (UCLA Anderson School of Management). Associations mining refers to the process of identifying associations (relationships) among data that appear frequently within the dataset. Classification, previously mentioned as classes definition, is the process of generalizing the known data, dividing it into classes, in order to then sort new data in the same structure. Clustering is the process which refers to the discovery of similar groups of data within the dataset, but without using previously known structures, as in Classification. Last but not least, sequential patterns mining looks for behavior patterns, and is often used for anticipation of certain trends (for example, predicting the likelihood of cheese being purchased in a transaction, based on a customer's purchase of bread and milk).

More data mining algorithms, techniques and relationship types exist, of course, that are not covered in this thesis.

2.3 Association Rules

2.3.1 Description

Association rule learning is a really well known and well researched method for the discovery of interesting relations among data in large datasets. Based on various interestingness measures, it identifies strong rules that apply on the data, which can provide useful information and knowledge. Critical decisions can be based on the results of this method, either in the business field, or in the medical science field, the sports field, the market field, and elsewhere.

The most frequent example used to explain the association rules mining concept is the market basket scenario, where a dataset containing information about many consumers' transactions exists and interesting relations among the transaction items are sought. Thus, if the association rule "{tomatoes, onions} => {milk}" is found with an acceptable percentage of interestingness, then the seller knows that a high percentage of customers usually buy milk when they buy tomatoes and onions as well.

The concept of association rules was popularised particularly due to the 1993 article of Agrawal et al., which has acquired more than 6000 citations according to Google Scholar, as of March 2008, and is thus one of the most cited papers in the Data Mining field. However, it is possible that what is now called "association rules" is similar to what appears in the 1966 paper on GUHA, a general data mining method developed by Petr Hájek et al [16].

2.3.2 Uses / applications

Association rule mining has been – and is still being – widely used in various areas, such as telecommunication networks, marketing, risk management, inventory control, betting calculation systems, casinos, sports, environmental research, medical science, and elsewhere.

2.3.3 Rules format

Typically, a rule follows a structure of the form X => Y, where X and Y are subsets of transaction items. The X => Y format means that if a transaction contains the items of the X item set, it is probable that it will also contain the items of the Y item set. No items that belong to the X item set can belong to the Y item set, and vice versa. The X item set is called “antecedent” or left-hand-side, LHS, while the Y item set is called “consequent” or right-hand-side, LHS.

The probability of a transaction containing items that belong to the antecedent item set to also contain items that belong to the consequent item set (in short, the probability of a rule to be applicable) depends on the interestingness measures on which basis the rule was found. Interestingness measures are covered in more detail in the next section.

2.3.4 Interestingness / significance measuring

Prior to searching for association rules in a given dataset, the interestingness measure has to be instructed. Simply put, the association rules mining algorithm must be told how “interesting” or significant the resulting rules should be. In other, more practical, words, the association rules mining algorithm must be told how *frequent* a pattern should be in order for it to be regarded as an association rule.

Three constraints are the most well-known on significance measuring: Minimum *Support*, minimum *Confidence*, and *Lift.*

The *Support* of a rule is considered to be: the proportion of the appearance of the items of that rule in the dataset’s transactions, together. For example, if the dataset consists of 10 transactions, 4 of which contain items X and Y, then the support of the rule “X => Y” (and the rule “Y => X” as well) would be 4 / 10, which equals 0.25 or 25%.

$$\text{supp}(X \Rightarrow Y) = \text{Transactions containing X, Y / Total transactions}$$

The *Confidence* of a rule is considered to be: the proportion of the appearance of the items of that rule in the dataset's transactions which contain the items of the LHS of the rule. For example, if the dataset consists of 10 transactions, 4 of which contain item X and only 2 of those containing item Y as well, then the confidence of the rule "X => Y" would be 2 / 4, which equals 0.5 or 50%.

$$conf(X => Y) = supp(X => Y) / supp(X)$$

The *lift* of a rule is considered to be: *the ratio of the observed support to that expected if X and Y were independent* [19].

Lift is a measure of the performance of a targeting model (association rule) at predicting or classifying cases as having an enhanced response (with respect to the population as a whole), measured against a random choice targeting model. A targeting model is doing a good job if the response within the target is much better than the average for the population as a whole. Lift is simply the ratio of these values: target response divided by average response [19].

The lift of a rule is defined as follows:

$$lift(X => Y) = supp(X => Y) / supp(X) * supp(Y)$$

As a general rule, when the lift of a rule is 1 then the items of that rule are, as is the terminology, *statistically independent*. In simple words, such a rule has no value, it is of no use. A rule's value increases as its lift increases.

2.3.5 The Apriori algorithm

One of the most popular frequent item set mining and association rule learning algorithms over transactional databases is the Apriori algorithm. Apriori is designed to operate on transactional databases, i.e. databases containing transactions. Each transaction is seen as a set of items, and Apriori defines the item sets that meet certain significance measures, with regards to their frequency of appearance inside the database.

Apriori works as follows: it first identifies the frequent individual item sets in the database, and then extends them to larger and even larger item sets, as long as these item sets appear sufficiently often in the database [15].

So Apriori uses a “bottom-up” approach – extending the frequent item sets, one at a time, and testing them against the data. The algorithm’s termination point is reached when no further transactions meeting the significance criteria are found.

2.3.6 Association rule mining in R - required packages

R does not natively possess any built-in function to enable the user to apply the Apriori algorithm over a set of transactional data. However, due to R’s great extensionality, packages have been built which are freely available to use and which provide the user with the means to execute the Apriori algorithm.

The packages required to execute the Apriori algorithm that are used in this thesis are the packages named “arules” and “arulesViz”.

The package “arules” contains all the necessary objects and structures for the inspection of the transactional item set, the execution of the Apriori algorithm, and the verification of the resulting data. Officially: Package “arules” *provides the infrastructure for representing, manipulating and analyzing transaction data and patterns (frequent itemsets and association rules). Also provides interfaces to C implementations of the association mining algorithms Apriori and Eclat by C. Borgelt* [6].

The “arules” package depends on the packages named “stats”, “methods” and “Matrix” (version 1.0-0 or greater). Dependency in R refers to the need of certain packages’ existence in the system prior to the installation of the package in question, in order for it to work correctly. So the user has to make sure that these three packages are already installed in the system before attempting to install “arules”.

Package “arules” also requires an installation of R version 2.14.2 or greater. Package “arules” version is currently 1.0-15 (November 2013).

The package “arulesViz” contains all the necessary objects and structures required for the visualization of the Apriori algorithm results. Using “arulesViz” the resulting association rules can be visualized in several visualization ways, including histograms, pie charts, dot plots, bar charts, line charts, box plots, even 3D scatterplots. Officially: Package “arulesViz” provides *various visualization techniques for association rules and itemsets. The packages also include several interactive visualizations for rule exploration. This package extends package arules* [7].

As stated in the official description of the package, “arulesViz” extends package “arules”, thus the first dependency that should be met in order for “arulesViz” to work correctly is the “arules” package (and specifically, “arules” version 1.0-5 or

greater). Other dependencies that have to be satisfied are the packages "scatterplot3d", "vcd", "seriation", and "igraph".

Finally, package "arulesViz" also requires an installation of R version 2.14.0 or greater. Package "arulesViz" version is currently 0.1-7 (November 2013).

2.3.7 Package installation in R

Installation of a package in R can be done in two ways: Either the package can be automatically downloaded from a package repository and installed, or it can be compiled by source code and then installed. Obviously, the first option is easier, less error-prone, and guarantees that the latest stable version of the package will be installed. On the downside, it depends on the availability of an internet connection.

To install a package by downloading it from a package repository, the function "install.packages()" needs to be used, thus:

```
> install.packages("<package.name>")
```

When issued, in most R versions this command will cause a dialog box to appear, prompting the user to select the repository mirror by which they wish to download the package. Alternatively, the internet address of the mirror can be manually specified by hardcoding it in the ".Rprofile" file:

```
r = getOption("repos") # hard code the UK repo for CRAN
r["CRAN"] = http://cran.uk.r-project.org
options(repos = r)
rm(r)
```

To install a package from its source code, the file name and path of the source code must be specified, the "type" parameter must be set to "source", and the "repos" parameter must be set to "NULL". Package source codes are usually compressed in "tar" files. For example:

```
> install.packages("/path/to/source.tar.gz", type="source",
+ repos=NULL)
```

An alternative way to install a package from its source code without using the R console at all is to use the "R CMD INSTALL" command on the respective operating system's command prompt. The "R CMD INSTALL" command needs only the filename and path of the source code file. The use is demonstrated below:

```
~$ R CMD INSTALL /path/to/source.tar.gz
```

2.3.8 Verification of a package installation in R

To verify that a package has been correctly installed in R, the functions “installed.packages()” and “library” are available, and suitable for the job.

The “installed.packages()” function will output a list containing every package that is installed in the current R installation. If the package in question is included in the list, its installation was successful.

The “library()” function is the function used to load a package into the memory, so as to begin using it. The best way of verifying the correct installation of a package is to attempt to load and use it.

To load a package, its name has to be specified in the “library()” function call, thus:

```
> library(“<package.name>”)
```

For example, to successfully load the packages “arules” and “arulesViz”, and thus verify their correct installation, the procedure that should be followed is the following (notice how many dependencies should be met on the “arulesViz” package):

```
> library(“arules”)

Loading required package: Matrix
Loading required package: lattice

Attaching package: ‘arules’

The following object is masked from ‘package:base’:

    %in%, write

Warning message:
package ‘arules’ was built under R version 3.0.2

> library(“arulesViz”)

Loading required package: scatterplot3d
Loading required package: vcd
Loading required package: grid
Loading required package: seriation
Loading required package: cluster
Loading required package: TSP
Loading required package: gclus
Loading required package: colorspace

Attaching package: ‘seriation’

The following object is masked from ‘package:lattice’:

    panel.lines

Loading required package: igraph
```

```
Attaching package: 'igraph'

The following object is masked from 'package:gclus':

    diameter


Attaching package: 'arulesViz'


The following object is masked from 'package:base'

    abbreviate

Warning messages:
1: package 'arulesViz' was built under R version 3.0.2
2: package 'vcd' was built under R version 3.0.2
3: package 'seriation' was built under R version 3.0.2
4: package 'TSP' was built under R version 3.0.2
5: package 'gclus' was built under R version 3.0.2
6: package 'colorspace' was built under R version 3.0.2
7: package 'igraph' was built under R version 3.0.2
```

It is notable that in this situation the 'library("arules")' command could be omitted, as the "arules" package is a dependency of the "arulesViz" package, and as such it would be automatically loaded upon the 'library("arulesViz")' command.

2.3.9 Using Apriori in R

After successfully installing the appropriate packages, R is set up and ready to start mining transactional datasets for association rules.

To mine association rules using the Apriori algorithm, the "arules" package provides a very useful function that combines the specification of almost every parameter that the Apriori algorithm needs to know, and "does it all": the "apriori()" function.

According to the "arules" package documentation [9], the "apriori()" function call consists of 4 parameters:

- data – an object of class "transactions" or any data structure which can be coerced into transactions (for example, a binary matrix or data.frame).
- parameter: object of class "APparameter" or named list. The default behavior is to mine rules with support 0.1, confidence 0.8, and maxlen 5.
- appearance: object of class APappearance or named list. With this argument item appearance can be restricted. By default all items can appear unrestricted.
- control: object of class APcontrol or named list. Controls the performance of the mining algorithm (item sorting, etc.)

The “data” parameter, in short, defines the dataset from which the association rules will be mined. The dataset has to be an object of class “transactions” or any data structure which can be coerced into transactions, and should generally follow two formats: Either the “basket” format, according to which all of the transaction’s items are included in one row and separated by a certain character, or the “single” format, according to which the transaction’s items are included in separate rows, which contain at least the transaction id. Officially:

For ‘basket’ format, each line in the transaction data file represents a transaction where the items (item labels) are separated by the characters specified by sep. For ‘single’ format, each line corresponds to a single item, containing at least ids for the transaction and the item [9].

The transactions are read from the dataset using the “read.transactions()” function, parameters of which are the “format” parameter mentioned earlier and the “sep” parameter, mentioned in the official formats description.

The “parameter” parameter accepts a list of parameters regarding the Apriori algorithm. This is the parameter where the interestingness / significance measures are defined. For example, to define a minimum support of 0.1, a minimum confidence of 0.8 and a maximum rule length of 5 (which defines that the total items of the resulting rules will be at most 5 per rule), the “parameter” parameter should be defined thus:

```
> apriori(dataset,parameter=list(sup=0.1,conf=0.9,minlen=2))
```

The “appearance” parameter instructs the algorithm about the desired appearance of the resulting association rules. For example, it might be desirable that only rules that contain “potatoe” in their LHS are mined. This can be achieved by setting the “appearance” parameter as follows:

```
> apriori(dataset,appearance=list(lhs=c("potatoe"),
+ default="rhs"))
```

The ‘default=”rhs”’ part instructs the algorithm that the RHS of the rule should be the default one (hence no constraints are to be applied to it).

Finally, the “control” parameter defines several parameters regarding the performance of the algorithm. Such parameters may refer to the resulting rules’ sorting order, the sorting algorithm to be used, memory usage and handling, and other control settings that need not be covered in this thesis. For example, to instruct the algorithm to optimize the memory usage, the “memopt” parameter should be set to TRUE:

```
> apriori(dataset,control=list(memopt=TRUE))
```

Alternatively, if the sorting algorithm used should be the “quicksort” algorithm, instead of the “heapsort” algorithm that is being used by default, then the “heap” parameter should be set to “FALSE”:

```
> apriori(dataset,control=list(heap=FALSE))
```

Thus, with just 4 parameters, the “apriori()” function offers a great amount control over the association rules mining.

Sometimes, rules with an empty LHS are produced. For example, a rule like the following might sometimes occur:

```
{} => {yogurt} 0.05603010  0.05603010 1.572722
```

This kind of rules show the probability of the items contained in their RHS to show up in a rule’s RHS, no matter what items exist in the LHS of that rule. So, obviously, this kind of rules’ support and confidence measures will be equal, and their lift measure will be 1.

The significance measures of the resulting association rules can be obtained using the “quality()” function:

```
> quality(rules)
```

```
     support confidence     lift
1 0.13941428 0.13941428 1.000000
2 0.05328452 0.05328452 1.000000
3 0.05603010 0.21925985 1.572722
4 0.02613382 0.10226821 1.919285
```

An important fact to note is that “apriori()” only produces rules with one item in their RHS.

2.3.10 Mining the rules

To mine a transactional dataset for interesting association rules, the general procedure that should be followed is:

- Loading of the dataset into R
- Pre-Processing of the dataset
- Mining of the rules

For the sakes of this example, a dataset included in the “arules” package will be used, which contains 9835 transactions of a groceries store. This dataset is stored in the “Groceries” object. Each row of the dataset is a transaction. For the sakes of simplicity, the data in this dataset is already in the “basket” format (described later

on). This example will attempt to mine rules from this dataset using the Apriori algorithm.

Before any action can be taken on a dataset, it has to be loaded into the memory. The “data()” function loads a dataset into the memory, as follows:

```
> data(<dataset.name>)
```

Thus, to load the Groceries dataset, the appropriate command would be:

```
> data(“Groceries”)
```

If no output is produced, the dataset was loaded successfully. Alternatively, if the dataset is stored in a database or a file, other functions are used to read such datasets, such as the “dbReadTable()” function or the “read.table()” function, respectively.

After loading the dataset, a pre-processing of the data is required in most cases (in some cases, a pre-processing of the data is required even before reading the dataset into R). The pre-processing of the data is a procedure where the dataset is edited appropriately in order for it to be compatible with the destination data mining system (in this case, the R language), and for the data to be valid for association rule mining (i.e. does not have missing values, non-categorical values are categorized / nominalized), so that the resulting rules are real and have a meaning.

In this example the pre-processing stage is omitted for the sakes of simplicity, since, as previously mentioned, the data is already in the “basket” format, and ready for association rule mining.

After the pre-processing is complete, it is time to mine the association rules. When the decisions regarding the significance measures are made, the “apriori()” function is ready to be called. In this example, the minimum support for a rule to be mined will be 0.08 and the minimum confidence will be 0.06:

```
> rules <-
+ apriori(Groceries,parameter=list(supp=0.08,conf=0.06))


parameter specification:
 confidence minval smax arem  aval originalSupport support
       0.06    0.1    1 none FALSE            TRUE    0.08
minlen maxlen target   ext
     1     10  rules FALSE

algorithmic control:
 filter tree heap memopt load sort verbose
    0.1 TRUE TRUE  FALSE TRUE    2    TRUE

apriori - find association rules with the apriori algorithm
version 4.21 (2004.05.09)        (c) 1996-2004   Christian
```

```
Borgelt
set item appearances ...[0 item(s)] done [0.00s].
set transactions ...[169 item(s), 9835 transaction(s)] done
[0.01s].
sorting and recoding items ... [13 item(s)] done [0.00s].
creating transaction tree ... done [0.00s].
checking subsets of size 1 2 done [0.00s].
writing ... [13 rule(s)] done [0.00s].
creating S4 object  ... done [0.00s].
```

With the given significance measures, 13 rules were found. They can be viewed via the use of the “inspect()” function, thus:

```
> inspect(rules)
```

```
   lhs    rhs                    support confidence lift
1  {}  => {bottled beer}     0.08052872 0.08052872    1
2  {}  => {pastry}           0.08896797 0.08896797    1
3  {}  => {citrus fruit}     0.08276563 0.08276563    1
4  {}  => {shopping bags}    0.09852567 0.09852567    1
5  {}  => {sausage}          0.09395018 0.09395018    1
6  {}  => {bottled water}    0.11052364 0.11052364    1
7  {}  => {tropical fruit}   0.10493137 0.10493137    1
8  {}  => {root vegetables}  0.10899847 0.10899847    1
9  {}  => {soda}             0.17437722 0.17437722    1
10 {}  => {yogurt}           0.13950178 0.13950178    1
11 {}  => {rolls/buns}       0.18393493 0.18393493    1
12 {}  => {other vegetables} 0.19349263 0.19349263    1
13 {}  => {whole milk}       0.25551601 0.25551601    1
```

Evaluating the rules, it can be concluded that the item “whole milk” is the most “popular” item, as it is found in the 25.55% of the transactions (because it has a support value of 0.25551601), which is equal to almost 2513 transactions, out of 9835. The second most popular item would be the “other vegetables” item, with a support of 0.19349263, therefore an appearance in the 19.35% of the transactions, equaling almost 1816 transactions.

Though much can be said about the resulting association rules, the result only contains rules with an empty LHS. To make the algorithm mine rules with at least one item in both sides, the “minlen” parameter should be set to “2”. Also, in order to mine some more rules, the minimum support of each rule will be decreased to 0.04:

```
> rules <-
+ apriori(Groceries,parameter=list(supp=0.04,conf=0.06,
+ minlen=2))
```

```
parameter specification:
 confidence minval smax arem  aval originalSupport support       0.06
0.1    1 none FALSE            TRUE    0.04
minlen maxlen target   ext
     2     10  rules FALSE
```

```
algorithmic control:
 filter tree heap memopt load sort verbose
    0.1 TRUE TRUE  FALSE TRUE    2    TRUE

apriori - find association rules with the apriori algorithm
version 4.21 (2004.05.09)        (c) 1996-2004   Christian
Borgelt
set item appearances ...[0 item(s)] done [0.00s].
set transactions ...[169 item(s), 9835 transaction(s)] done
[0.00s].
sorting and recoding items ... [32 item(s)] done [0.00s].
creating transaction tree ... done [0.00s].
checking subsets of size 1 2 3 done [0.00s].
writing ... [18 rule(s)] done [0.00s].
creating S4 object  ... done [0.00s].
```

In this case, 18 rules were mined:

```
   lhs                    rhs                    support confidence
lift
1  {tropical fruit}    => {whole milk}       0.04229792  0.4031008
1.5775950
2  {whole milk}        => {tropical fruit}   0.04229792  0.1655392
1.5775950
3  {root vegetables}   => {other vegetables} 0.04738180  0.4347015
2.2466049
4  {other vegetables}  => {root vegetables}  0.04738180  0.2448765
2.2466049
5  {root vegetables}   => {whole milk}       0.04890696  0.4486940
1.7560310
6  {whole milk}        => {root vegetables}  0.04890696  0.1914047
1.7560310
7  {soda}              => {whole milk}       0.04006101  0.2297376
0.8991124
8  {whole milk}        => {soda}             0.04006101  0.1567847
0.8991124
9  {yogurt}            => {other vegetables} 0.04341637  0.3112245
1.6084566
10 {other vegetables}  => {yogurt}           0.04341637  0.2243826
1.6084566
11 {yogurt}            => {whole milk}       0.05602440  0.4016035
1.5717351
12 {whole milk}        => {yogurt}           0.05602440  0.2192598
1.5717351
13 {rolls/buns}        => {other vegetables} 0.04260295  0.2316197
1.1970465
14 {other vegetables}  => {rolls/buns}       0.04260295
0.2201787 1.1970465
15 {rolls/buns}        => {whole milk}       0.05663447  0.3079049
1.2050318
16 {whole milk}        => {rolls/buns}       0.05663447  0.2216474
1.2050318
17 {other vegetables}  => {whole milk}       0.07483477  0.3867578
1.5136341
18 {whole milk}        => {other vegetables} 0.07483477  0.2928770
1.5136341
```

Observing the rules, it is easily perceived that the most frequent item set is the {whole milk, other vegetables} item set, with a support of 0.07483477. Between the two, the item that has the highest probability to be included in a transaction if the other is included as well is the ”whole milk” item, since the “{other vegetables} => {whole milk}” rule has a confidence value of 0.3867578, which is greater than the confidence value of the “{whole milk} => {other vegetables}” (0.2928770). Therefore, it is more probable that a transaction containing “other vegetables” will also contain “whole milk” than a transaction containing “whole milk” to also contain “other vegetables”.

2.3.11 Visualizing the results

Apart from extracting the rules in text form, a really useful feature that R offers is the visualization of the resulting association rules, via various graphics. As it has already been mentioned, rules can be visualized in histograms, bar charts, pie charts, line charts, dot plots, even 3D diagrams. Visualization of association rules can be very mind-opening and clarifying, since a visual representation of the rules can help pinpoint several aspects that were difficult to see before.

To perform any kind of association rules visualization, the “arulesViz” package is required. The procedure of obtaining, installing and loading “arulesViz” is described in chapter 2.3.7 “Package installation in R”.

Once “arulesViz” has been successfully installed, R is ready to begin visualizing the association rules. To begin, the “arulesViz” package should be loaded into the memory, via the “library()” function:

```
> library(“arulesViz”)

[output omitted]
```

The simplest visualization that can be performed is a simple scatter plot. Its x-axis will represent the support of the rules, while its y-axis will represent the confidence of the rule.

To plot a scatterplot, the “plot()” function is used, and it is called as follows:

```
> plot(<rules.object>)
```

Thus, continuing the previous example, the “plot()” function will be used to create a simple scatterplot of the resulting rules. To clarify, the “rules” object contains 18 rules having a minimum support of 0.04, a minimum confidence of 0.05, and a minimum length of 2. The plotting procedure is the following:

```
> plot(rules)
```

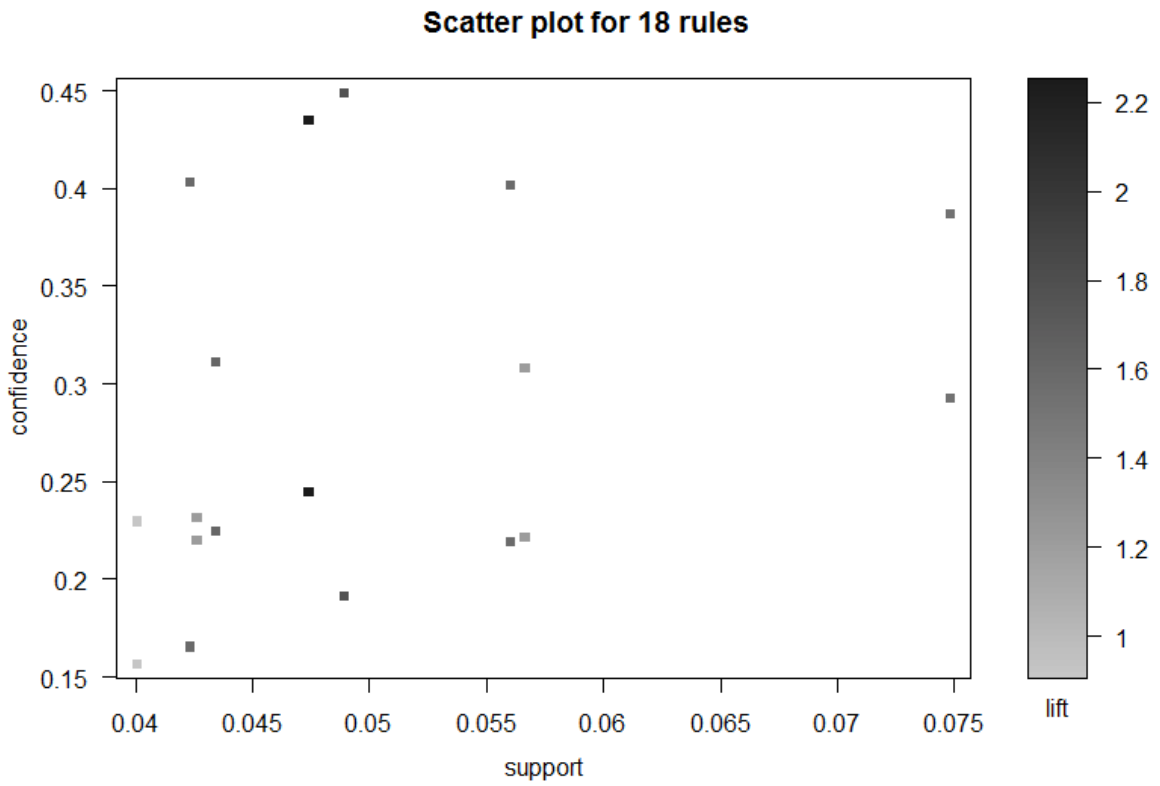

Figure 2.3.11-1: Visual representation of rules via scatter plot

A scatter plot has been created. Observing it, an extra bar can be seen to the right, which contains the lift colorization. According to this bar, each point in the scatter plot is colored according to is lift value, therefore each point can represent 3 different significance measures at the same time – in this case, support, confidence and lift.

To create a scatterplot and choose the measures that the x and y axes will represent, the “measure” parameter has to be defined. If, for example, the desired measure for the x-axis is “confidence”, and the desired measure for the y-axis is “lift”, then the plot function needs to be called thus:

```
> plot(rules,measure=c(“confidence”,”lift”))
```

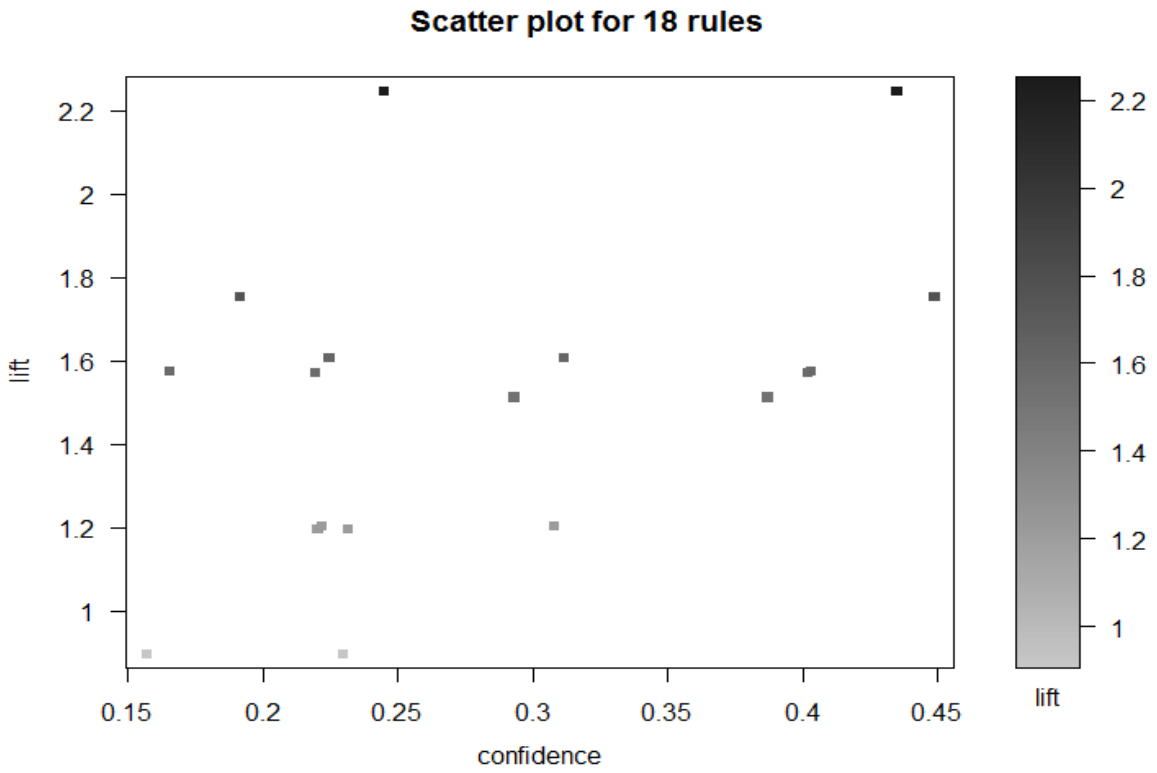

Figure 2.3.11-2: Visual representation of rules via scatter plot

Observing the newly created scatter plot, though it is easily perceived that the x and y axes measures are indeed “confidence” and “lift”, the lift measure is still present in the color bar to the right. As the “lift” measure is already represented by the y-axis, its color representation as well is exaggeration. To replace the “lift” measure in the color bar with, for example, “support”, the “shading” parameter must be adjusted:

```
> plot(rules,measure=c("confidence","lift"),shading="support")
```

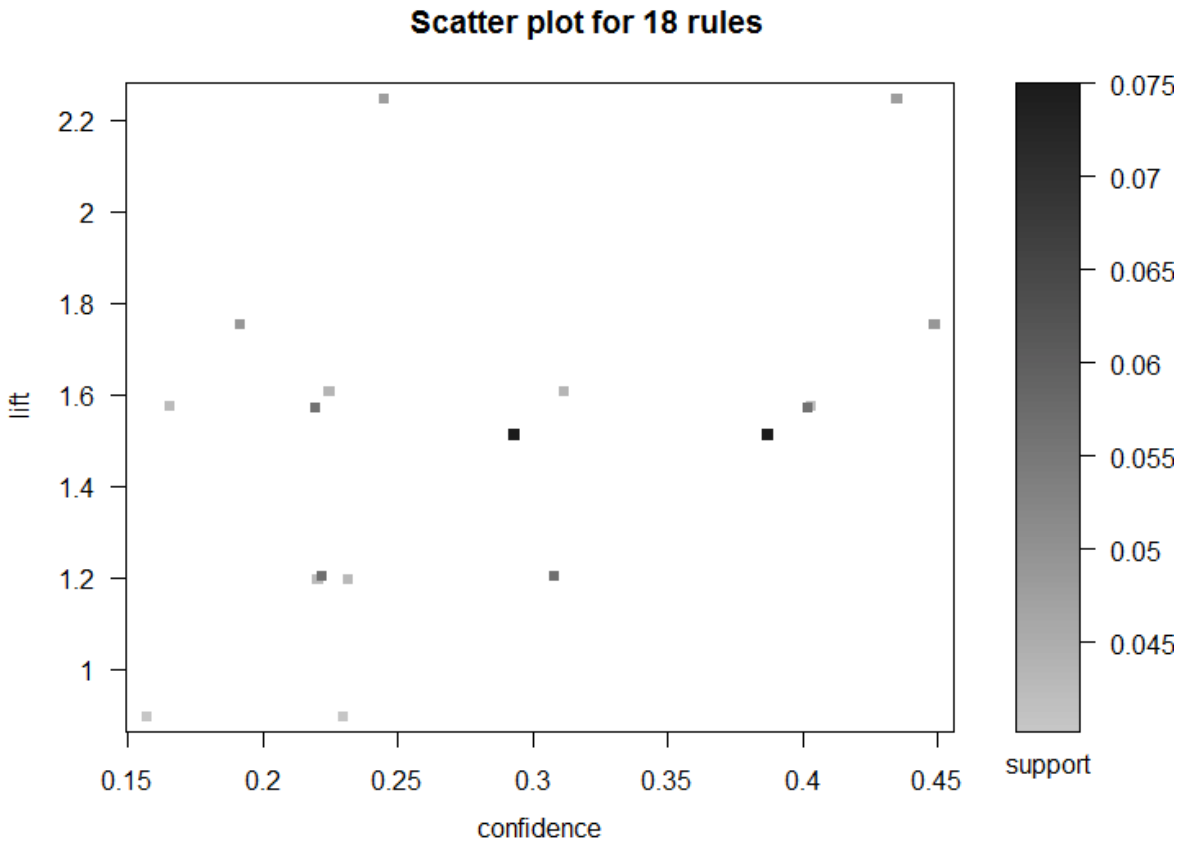

Figure 2.3.11-3: Visual representation of rules via scatter plot

Now the rules are colorized according to their support rather than their lift.

Apart from the simple scatter plot, various other visualization graphs are available in the “arulesViz” backage. For example, the rules can be visualized as a matrix:

```
> plot(rules,method="matrix")

Itemsets in Antecedent (LHS)
[1] "{tropical fruit}"   "{whole milk}"       "{root
vegetables}"  "{other vegetables}" "{soda}"
"{yogurt}"           "{rolls/buns}"
Itemsets in Consequent (RHS)
[1] "{whole milk}"       "{tropical fruit}"   "{other
vegetables}" "{root vegetables}"  "{soda}"
"{yogurt}"           "{rolls/buns}"
```

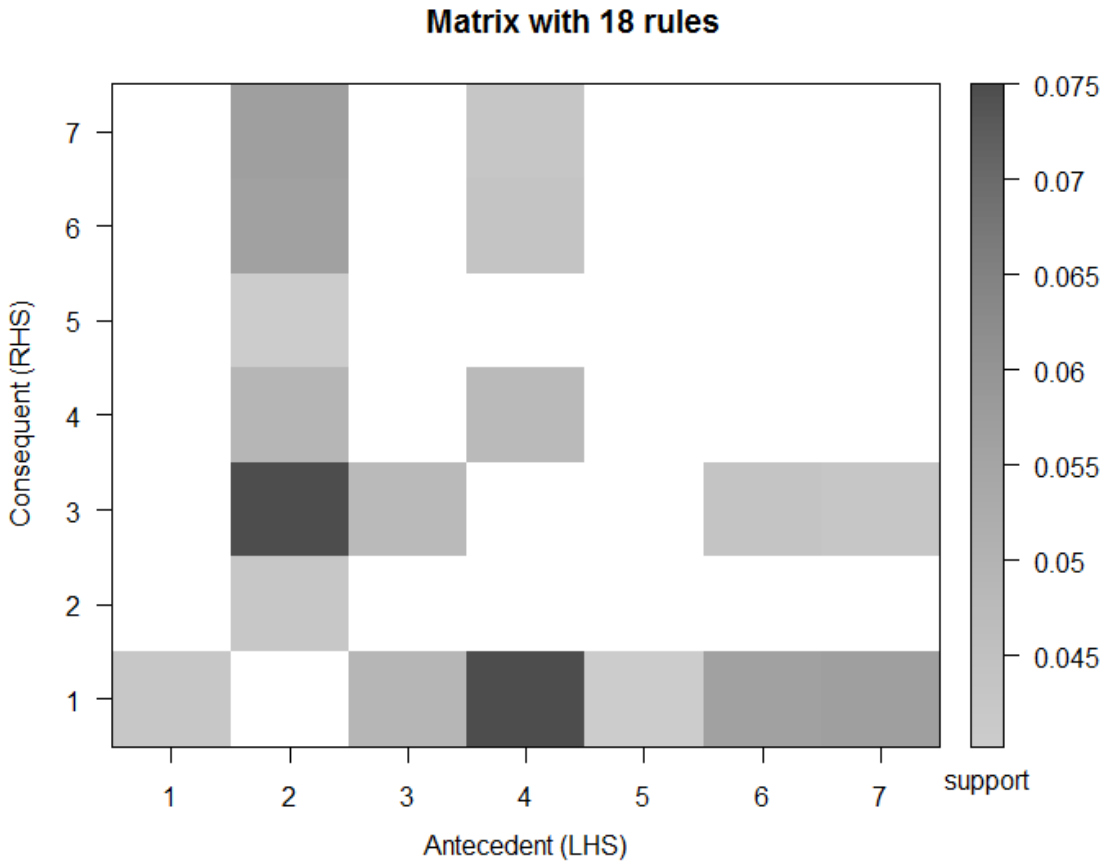

Figure 2.3.11-4: Visual representation of rules via matrix

The matrix graphic plots the antecedent and consequent of the rules. Each axis is split into 7 equal parts (1 for each different item contained in the rules set, the item order is output in the console), thus the matrix consists of 49 equal parts which represent the rules. Each rule is colorized by its support measure, as seen in the color bar to the right of the matrix. It is important to note that the numbers in the axes represent different items in each axis – for example, number 4 in the x-axis could stand for {whole milk} while number 4 in the y-axis could stand for {root vegetables}.

One great feature of the “plot()” function is the “interactive” parameter, which, if set to TRUE, allows the user to use the mouse and click various points of the graphic, each click causing information about the clicked rule to be displayed.

The rules can be also visualized as groups, using the “grouped” method:

```
> plot(rules,method="grouped")
```

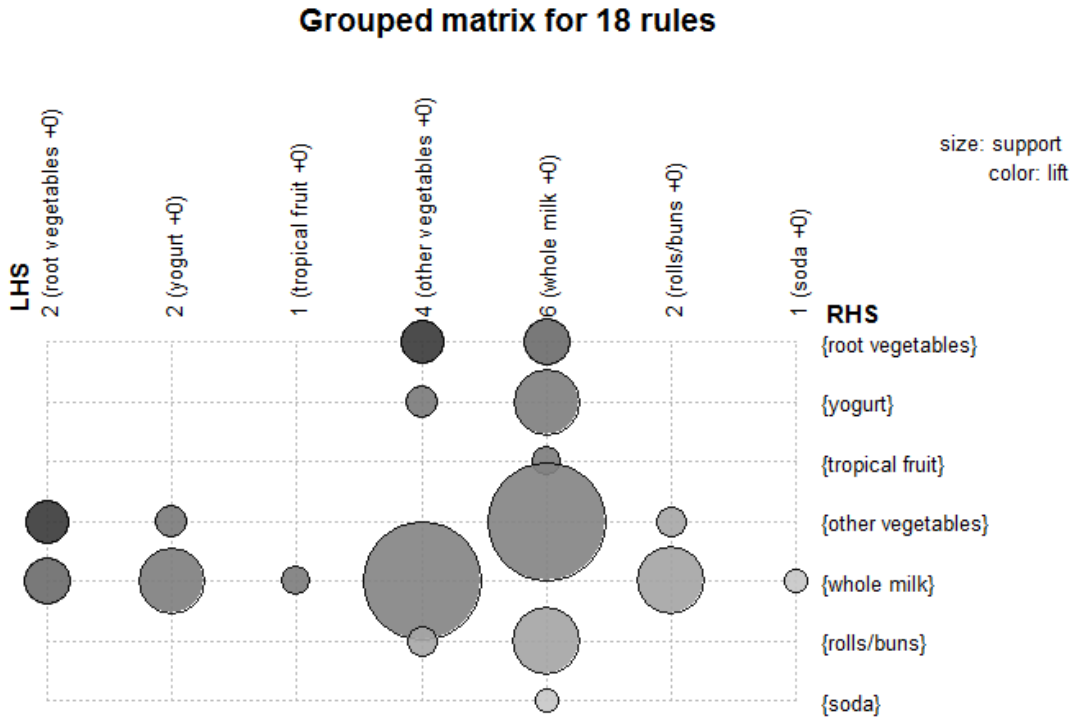

Figure 2.3.11-5: Visual representation of rules via groups

The “grouped” method generates a group graph, where each rule is presented by a circle (or disk) as a group (similarly to mathematical groups). The size of the circle/disc represents the rule’s support value, while the circle/disc color represents the rule’s lift (the represented measures can be seen in the top-right corner of the graph).

Finally, the “matrix3D” method will be demonstrated, where the rules are represented in a three-dimensional graph:

```
> plot(rules,method=”matrix3d”)

Itemsets in Antecedent (LHS)
[1] "{tropical fruit}"   "{whole milk}"       "{root
vegetables}"  "{other vegetables}" "{soda}"
"{yogurt}"           "{rolls/buns}"
Itemsets in Consequent (RHS)
[1] "{whole milk}"       "{tropical fruit}"   "{other
vegetables}" "{root vegetables}"  "{soda}"
"{yogurt}"           "{rolls/buns}"
```

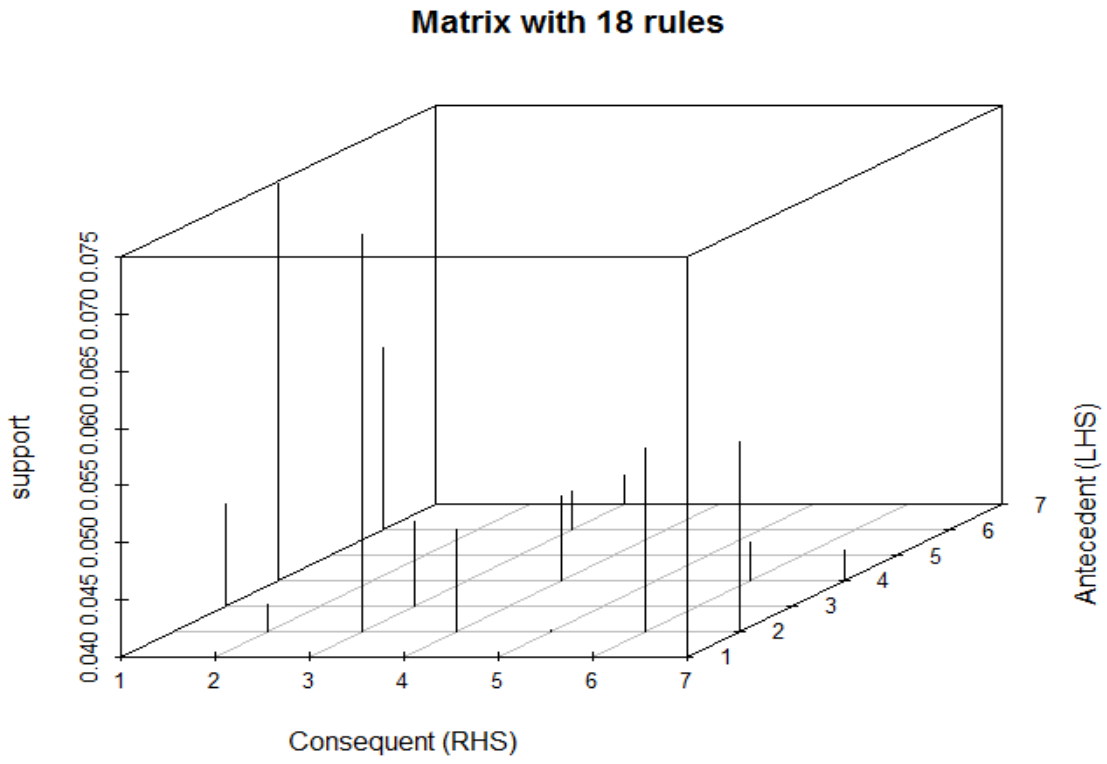

Figure 2.3.11-6: Visual representation of rules via 3D matrix

The “matrix3d” method creates a three-dimensional rectangle. The x and y axes once again represent the antecedent and consequent of the rules, while the z-axis represents the support of each rule. The “measure” parameter can be used in the “matrix3d” graphic as well to adjust the measure being represented by the z-axis.

As a conclusion, it is important to note that all of the plotting methods can be modified and adjusted at will, using the “measure” and “shading” parameters as previously described.

2.3.12 Storing the results

Once having successfully mined the association rules, the next prudent step one would take is to store them somewhere, in order for them to be available after the current R session has ended. The possible storage solutions vary: one could either save the results to a file, or export them to another statistical analysis system, or store them on a remote server over a network connection, or insert them into a database. In this thesis, the approaches covered are exporting the resulting rules to a file, and writing them to a database.

To write the results in a database, a connection to the database has to be made by R, and then the appropriate queries must be submitted to the database by R through this connection, in order for the rules to be stored as desired.

Communicating with a database from R will be covered in more detail in the next chapter.

To save the resulting rules to a file, the typical procedure that should be followed is:

- Conversion of the rules to a format appropriate for exporting to a file
- Creating the file

When exporting the rules, the resulting object is an object of the “rules” class, which is an “S4” class, part of the “arules” package. The goal is for this object to be converted to a format suitable for exporting to a file – a data fame, for example.

To convert (or, to use R’s terminology, “coerce”) a “rules” object to a data frame, a function from the “as” family has to be used. The “as” functions family is a set of functions used for converting (“or coercing”) objects from one type to another. Usually, the name of the function implies the returned object type – for example, the “as.factor()” function will convert / coerce an object into a factor (and return it, of course).

Hence, to convert a “rules” object to a data frame, the function that must be used is the “as()” function. It is important at this point to mention that “S4” class objects (like the ones originating from the “arules” package) may not comply with the “as()” family functions, and can be successfully converted only by using the “as()” function, specifying the target object type as a parameter to it. Therefore, the appropriate function call to convert a “rules” object to a data frame would be the following:

```
> as(<rules.object>,”data.frame”)
```

Consequently, continuing the previous example, to convert the resulting association rules to a data frame, the appropriate command to be issued is:

```
> df <- as(rules,”data.frame”)
> df
```

```
                      rules    support confidence lift
1      {} => {bottled beer} 0.08052872 0.08052872    1
2            {} => {pastry} 0.08896797 0.08896797    1
3      {} => {citrus fruit} 0.08276563 0.08276563    1
4     {} => {shopping bags} 0.09852567 0.09852567    1
5           {} => {sausage} 0.09395018 0.09395018    1
6     {} => {bottled water} 0.11052364 0.11052364    1
7    {} => {tropical fruit} 0.10493137 0.10493137    1
8   {} => {root vegetables} 0.10899847 0.10899847    1
9              {} => {soda} 0.17437722 0.17437722    1
```

```
10           {} => {yogurt} 0.13950178 0.13950178    1
11       {} => {rolls/buns} 0.18393493 0.18393493    1
12 {} => {other vegetables} 0.19349263 0.19349263    1
13       {} => {whole milk} 0.25551601 0.25551601    1
```

The resulting rules have now been stored into the “df” object, which is a data frame:

```
> class(df)
[1] "data.frame"
```

The association rules are now ready to be written into a file. For maximum compatibility, the rules will be stored in a csv file. The function that will accomplish this is the “write.table()” function, the parameters that need to be specified are the “file” parameter – indicates the destination file – and the “sep” parameter – defines the delimiter character that the function will use to separate entries. An extra parameter that may be useful to specify is the “row.names” parameter, which, if set to TRUE, adds an extra column to the output file which contains the row names – a feature that may not be desirable in many cases as this column is treated like any of the rest, thus the resulting file will have one column more than intended. Finally, the appropriate command required is the following:

```
> write.table(df,file=”targetfile.csv”,sep=”,”)
```

2.4 Classification

2.4.1 Description

Another widely used method in the data mining field is the well-known “classification” method. Classification refers to the process of automatically building a model that can classify a class of objects, based on an already classified set of data, which can then correctly classify future objects according to a “class attribute”. A class attribute is the attribute on which the classification of the data will be based. In other words, it could be said that classification is a process of data generalization, according to various criteria.

Consequently, the classification method is a two-phase procedure:

During the first phase, a set of data is used to build a classification model. This set of data is already correctly classified according to a desired attribute, and as such it is used by the model in order for it to “learn how to classify”. Such a dataset is called a “training dataset”, and such a process is called “supervised learning”.

During the second phase, new, unknown data is provided to the classification model, and the model tries to correctly classify this data using the knowledge acquired by the classification of the training dataset.

To make things clearer, a short example shall be demonstrated:

Supposedly, it is the desire of a doctor to build a classification model which correctly diagnoses what the symptoms of various patients mean, in order to define the malady which they have been afflicted by. In order to build such a model, the doctor will need to supply a training dataset to train the model, which in this case can be a dataset containing records of past patients whose symptoms were correctly identified and diagnosed. Once the model has been trained, the doctor can begin using it to classify new patients.

It is very important at this point to note that, although the classification models are trained using correctly classified data, the classification of the new data is nevertheless not always 100% correct, since it is merely based on the statistical outcome of the training data set. Therefore, a classification model should never be the only clue which a decision is based on, especially in such cases that involve medical risk and as such cannot afford an incorrect classification / diagnosis. Unless, of course, it is the only clue available – when trying to predict unknown, future behaviors, for example.

2.4.2 A few words about supervised learning

Supervised learning is the machine learning task of inferring a function from labeled training data. The training data consist of a set of training examples. In supervised learning, each example is a pair consisting of an input object (typically a vector) and a desired output value (also called the supervisory signal). A supervised learning algorithm analyzes the training data and produces an inferred function, which can be used for mapping new examples. An optimal scenario will allow for the algorithm to correctly determine the class labels for unseen instances. This requires the learning algorithm to generalize from the training data to unseen situations in a "reasonable" way [25].

2.4.3 Uses / applications

Classification is one of the most widely used data mining methods in business, science, medical, and other major fields, for behaviors predicting, marketing trends, treatment effectiveness analysis, etcetera.

2.4.4 Classification versus prediction

It is important to distinguish the term “classification” from the term “prediction”. Classification refers to the categorization of existing, known data, data that it is desirable that it be categorized. Prediction, on the other hand, refers to the production of data that is unknown, based on a classification model, in order to predict future trends – for example, the outcome of a patient’s treatment.

It is not inadvisable that a classification model is used for the prediction of future data, but it is important to keep in mind the difference of classifying existing data and predicting new data, using the same classification model.

2.4.5 The C4.5 algorithm

Classifier constructing systems are among the commonly used tools in data mining. Such systems accept a collection of cases as an input, where each case belongs to a class and is described by its values for a pre-defined set of attributes, and produce a classifier which can accurately predict the class to which a new case belongs. The “C4.5” algorithm generates classifiers expressed as decision trees - though it can also construct classifiers in more comprehensible rule set form [4].

Decision trees are trees that classify cases by sorting them based on their attributes’ values. Each node of the tree contains an attribute, each branch of the node represents the possible outcomes of this attribute, and each leaf represents the decision taken about the case being examined after all attributes have been computed. For example, if the attribute selected is “Unemployed”, and the possible outcomes of this attribute are “Yes” and “No”, a node will be created for each outcome, which will contain the name of the next attribute to be processed. The procedure will then be repeated for that attribute, as many times as the different outcomes of the previous node attribute (thus, in this case, 2 times – one for the “Yes” outcome and one for the “No” outcome). If there are no more attributes to be processed, a leaf is created, containing the final decision made for the case. Nodes and leaves are also called decision nodes and leaf nodes, respectively.

2.4.6 Classification in R – required packages

Unfortunately, R does not have any built-in implementation for performing data classification. Thus, to perform data classification in R, extension packages need to be installed.

For the sakes of this thesis's examples and case studies, the packages "RWeka" and "party" were used. The package "RWeka" was built by the "Weka" project team, and provides an R interface to Weka, via which Weka's implementation of the C4.5 algorithm – called J48, which is the C4.5 algorithm implementation in Java – is available to R. Weka is a collection of machine learning algorithms for data mining tasks written in Java, containing tools for data pre-processing, classification, regression, clustering, association rules, and visualization.

To install the packages mentioned above, the following commands need to be issued:

```
> install.packages("RWeka")
> install.packages("party")
```

Package installation in R is described in more detail in chapter 2.3.7 "Package installation in R".

2.4.7 Using the C4.5 algorithm in R

After successfully installing the packages, they need to be loaded into R using the "library()" function:

```
> library("RWeka")
> library("party")
```

The procedure of loading a package into R and the usage of the "library()" function are described in more detail in chapter 2.3.8 "Verification of a package installation in R".

Once successfully loading the packages, R is ready to begin the classification of a supplied dataset. For the sakes of this example, the dataset 'IRIS' from package 'datasets' will be used. It consists of 50 objects from each of three species of Iris flowers (Setosa, Virginica and Versicolor). For each object four attributes are measured length and width of sepal and petal. The "iris" dataset is contained in a data frame object, which structure consists of four columns of class "numeric", which contain the observations for the Iris flowers' petals and sepals, and one column of class "factor", which contains the Iris species each flower belongs to.

2.4.8 Building a decision tree

To build a decision tree using the C4.5 / J48 algorithm, the "J48()" function should be used. The function's parameters are: "formula" - a symbolic description of the model to be fit (in most cases, this consists of a factor which contains the values

into which the dataset objects will be classified, and the names of the attributes which the dataset objects will be classified by), “data” – an optional data frame containing the variables in the model (in most cases, this is the dataset that contains the data to be classified, or a training data set), “subset” - an optional vector specifying a subset of observations to be used in the fitting process, “na.action” - a function which indicates what should happen when the data contain NAs, “control” - an object of class Weka_control giving options to be passed to the Weka learner, and “options” - a named list of further options, or NULL (default).

The use of the “J48()” function to build a decision tree for the “iris” dataset using the default parameters is demonstrated bellow:

```
> m1 <- J48(Species~., data=iris)
```

The function is called setting the “data” parameter to “iris”, thus instructing the algorithm that the dataset to classify is the “iris” dataset, and the “formula” parameter as “Species~.”, thus instructing the algorithm that the attribute that is desired to be predicted is the “Species” attribute, that the final outcome should be a categorization of the dataset objects by Species.

It is also important to note the use of the “~.” characters in the “formula” parameter. The formula parameter, as mentioned before, instructs the algorithm about two different things: the first is the attribute which the objects will be classified as, and the second is the attributes which the objects will be classified by. These two attribute sets are separated by the “~” (tilde) character: The classification result attribute factor is placed before the tilde, and the classification attributes for the algorithm to take into account are placed after the tilde, separated using the “+” (plus) character. For example, if it was desirable that the objects were classified only regarding their sepal width and length, then the appropriate function call would be:

```
> m1 <- J48(Species ~ Sepal.Length + Sepal.Width, data=iris)
```

The rest of the “J48()” function parameters can be specified at will, depending on the desirable outcome of the algorithm. For example, the “control” parameter accepts a “Weka_control” object, which can set any of the options that Weka provides when building a decision tree. Some of the available options are the “M” option, which specifies the minimum number of instances per leaf that should be met, or the “U” option, which specifies whether the resulting tree will be unpruned. Two more options that can be useful when building a decision tree are the “t” option, which is used to specify a training dataset to be used (in “arff” format), and the “T” option, which is used to specify some test data to be used (in “arff” format).

The above control options are passed to the “J48()” function as follows:

```
> m1 <- J48(Species~., data=iris,
+ control=Weka_control(M=2,U=T))
```

2.4.9 Pruning a decision tree

One of the questions that arise in a decision tree algorithm is the optimal size of the final tree. A tree that is too large takes the risk of overfitting the training data and thus poorly generalizing the new data. On the other hand, a tree that is too small may miss important structural information about the sample space. However, it is hard to define the point at which a decision tree algorithm should stop, because it is impossible to predict if the addition of one extra node will dramatically decrease error. This problem is known as the horizon effect. A common workaround is to let the tree grow until each node contains a small number of instances, and then use pruning to remove nodes that do not provide additional information.

Pruning is a technique in machine learning that reduces the size of decision trees by removing sections of the tree that provide little power to classify instances. The dual goal of pruning is reduced complexity of the final classifier as well as better predictive accuracy by the reduction of overfitting and removal of sections of a classifier that may be based on noisy or erroneous data [22].

In short, pruning a decision tree refers to the process of removing nodes and leaves of the tree that provide little or incorrect information – or no information at all. Pruning is based on certain criteria that the about-to-be-pruned nodes or leaves do not meet, such as the number of instances they hold (a node / leaf will not be partitioned further if the number of instances it holds is below a certain threshold), or their purity (a node / leaf will not be partitioned further if all induced splittings yield no significant impurity reduction). These criteria, though, need to be chosen very carefully, as a really low minimum of allowed instances per node / leaf will result in an oversized tree, whereas a really high threshold will cause the algorithm to omit useful splittings.

In R, to instruct the algorithm to apply pruning on a tree, the “U” parameter (“Unpruned”) must be set to true. “U” is a Weka parameter, therefore it needs to be placed in the “control” parameter of the algorithm, thus:

```
> m1 <- J48(Species~., data=iris, control=Weka_control(U=T))
```

Other (Weka) parameters devoted to the pruning control of the tree are the “C” parameter (“Confidence”), which is used to set a confidence threshold for pruning, or the “R” parameter (“Reduced”), which is used for reduced error pruning.

In general, pruning should reduce the size of a decision tree without reducing predictive accuracy as measured by a test set or using cross-validation.

2.4.10 Visualizing the results

After successfully building a decision tree using the “J48()” function, the resulting tree can be visually represented either in text form, where the tree structure is represented only by text, or in a graph form, where the tree is represented by a typical node graph.

The tree can be visualized in text form by its R object name:

```
> m1 <- J48(Species~., data=iris, control=Weka_control(U=T))
> m1

J48 unpruned tree
------------------

Petal.Width <= 0.6: setosa (50.0)
Petal.Width > 0.6
|   Petal.Width <= 1.7
|   |   Petal.Length <= 4.9: versicolor (48.0/1.0)
|   |   Petal.Length > 4.9
|   |   |   Petal.Width <= 1.5: virginica (3.0)
|   |   |   Petal.Width > 1.5: versicolor (3.0/1.0)
|   Petal.Width > 1.7: virginica (46.0/1.0)

Number of Leaves  :     5

Size of the tree :      9
```

As is easily perceived by this output, the root node of the tree is the “Petal.Width” attribute. All observations whose “Petal.Width” attribute value is less than or equal to 0.6 are classified as “setosa”. The rest are divided further, again by their “Petal.Width” – the ones that have a “Petal.Width” attribute value greater than 1.7 are classified as “virginica” and the rest are once more divided, this time by their “Petal.Length” attribute. The ones with a “Petal.Width” attribute value less than 4.9 are classified as “versicolor” and the rest are divided for the final time according to their “Petal.Width” attribute once again – the ones with a value less than or equal to 1.5 are classified as “virginica”, and the ones with a value greater than 1.5 are classified as “versicolor”.

The resulting leaves are 5 in total, which means that the dataset observations can be grouped into five different “teams”, the attributes of each team having certain similarities with each other – for example, similar petal widths, or similar sepal lengths.

The attribute that was chosen by the algorithm to be the root node of the tree is the “Petal.Width” attribute. The reason behind this choice is found in what is called *information gain*.

Information gain is a concept that measures the amount of information contained in a set of data. It gives the idea of importance of an attribute in a dataset. It is the mathematical tool that algorithm J48 uses to decide which variable fits better in each tree node, in terms of target variable prediction.

The information gain calculation will answer the question of why the algorithm has decided to start with attribute “Petal.Width”. Information gain is calculated by the function “information.gain()”, included in the “FSelector” package:

```
> library("FSelector")
> information.gain(Species~.,data=iris)

             attr_importance
Sepal.Length       0.6522837
Sepal.Width        0.3855963
Petal.Length       1.3565450
Petal.Width        1.3784027
```

From the above output, it is easily perceived that the “Petal.Width” attribute is the one that has the highest information gain (1.3784027), and as such it is chosen by the J48 algorithm to be the root node of the decision tree.

The same process is performed for each of the tree nodes. For example, once the root node has been set, every observation with a “Petal.Width” attribute value less than or equal to 0.6 is classified as “setosa” and the rest of the observations are to be further divided. Thus, the calculation of the second tree node is performed for a subset of the dataset – the one which includes only the observations that have not yet been classified:

```
> subset <- subset(iris, Petal.Width > 0.6)
> information.gain(Species~., data = subset)

             attr_importance
Sepal.Length       0.1605000
Sepal.Width        0.0000000
Petal.Length       0.6573738
Petal.Width        0.6901604
```

Observing the output above, it can be concluded that the “Petal.Width” attribute has again the highest information gain, and as such it will be the attribute chosen to inhabit the second tree node.

Further on, to view the classification details and the confusion matrix of the constructed tree, the “summary()” function must be used:

```
> summary(m1)

=== Summary ===

Correctly Classified Instances         147               98
%
Incorrectly Classified Instances         3                2       %
Kappa statistic                          0.97
Mean absolute error                      0.0233
Root mean squared error                  0.108
Relative absolute error                  5.2482 %
Root relative squared error             22.9089 %
Coverage of cases (0.95 level)          98.6667 %
Mean rel. region size (0.95 level)      34      %
Total Number of Instances               150

=== Confusion Matrix ===

  a  b  c   <-- classified as
 50  0  0 |  a = setosa
  0 49  1 |  b = versicolor
  0  2 48 |  c = virginica
```

The confusion matrix (also known as a contingency table or an error matrix) of a decision tree is a visual representation of the algorithm's performance in classifying the observations in the dataset. Its columns represent the classes into which the instances were classified, while its rows represent the classes to which the instances actually belong. Observation of the confusion matrix of a decision tree can provide knowledge about the accuracy of the algorithm's classification. For instance, observing the above output, it is perceivable that all of the observations that belong to the "setosa" class (represented by "a") were correctly classified as belonging to that class, whereas the observations belonging to the "versicolor" class (represented by "b") were not all classified correctly – one observation was classified as belonging to the "virginica" class. Similarly, two of the observations that belong to the "virginica" class (represented my "c") were incorrectly classified as belonging to the "versicolor" class. Thus, the error percentage of the algorithm in this decision tree construction was 3/150 = 0.02 = 2%.

To visualize the resulting tree in a graph form, the "plot()" function needs be used. There are two types which the tree can be visualized as, specified by the "type" parameter – the "simple" type, which does not display all of the resulting leaves' details, and the "extended" type, which displays every detail and is the default setting. The "plot()" function needs to be used thus:

```
> plot(m1)
```

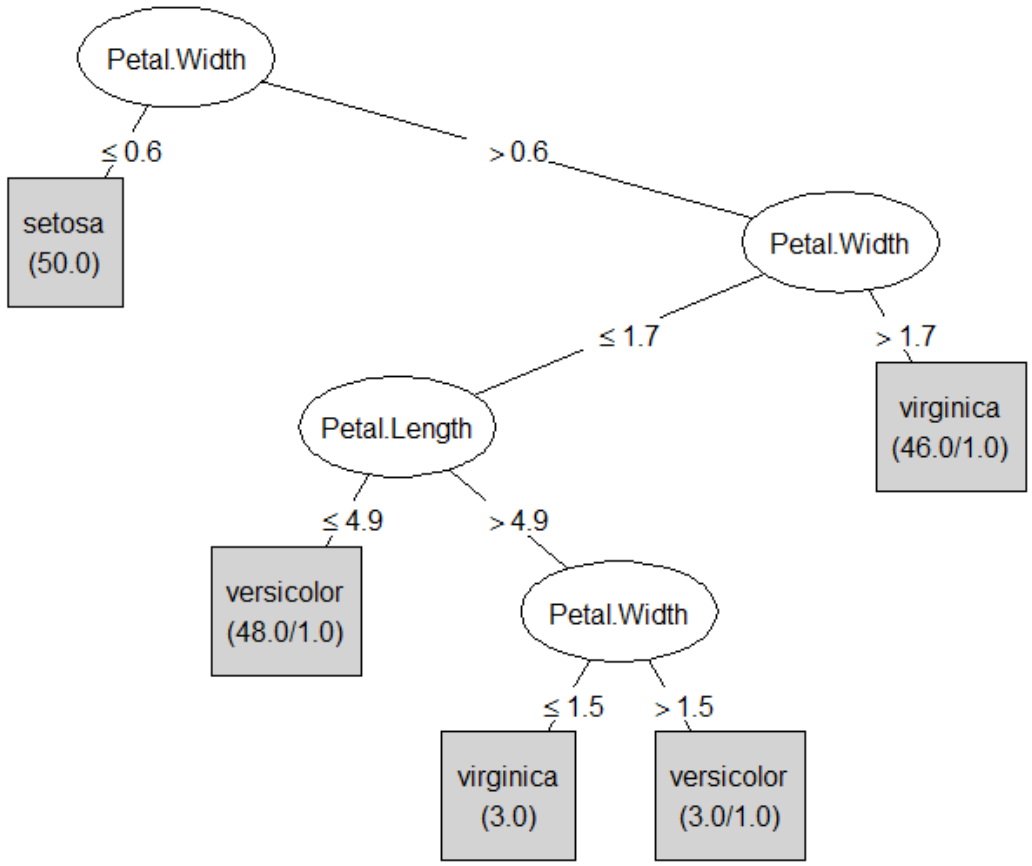

Figure 2.4.10-1: Visual representation of the decision tree

2.5 Conclusion

In this chapter the data mining scope of R was introduced and detailed. Taking advantage of the wide extensionality possibilities range of R, the installation and use of external packages was described, along with two data mining examples. Clearly, R's data mining capabilities are vast – with just three or four commands it is possible to mine association rules or build decision trees. The storage location of the resulting association rules and decision trees, except from files, can also be a database – an option which provides more security, contributes in the data availability, and is generally preferable to storing data mining results in files. R's communication capabilities with a DBMS are described in the next chapter.

3. R to DBMS Connectivity

3.1 Introduction

A DBMS – Database Management System – is a specifically designed application to allow a user to manipulate and interact with a database. According to Wikipedia:

Database management systems (DBMSs) are specially designed applications that interact with the user, other applications, and the database itself to capture and analyze data. A general-purpose database management system (DBMS) is a software system designed to allow the definition, creation, querying, update, and administration of databases [18].

Though DBMSs are very powerful and offer lots of features and database management tools, a typical DBMS has very limited numerical and statistical features. Metrics like quantiles and medians, easily calculated in a statistical analysis system, require complex and sophisticated SQL queries in order to be calculated in a DBMS. What is more, the numerical algorithms that use the basic SQL aggregate functions in order to perform statistical tasks do not have implementations to ensure numerical accuracy, and the wide range of the SQL data types may cause unexpected numerical roundings and errors when converting numbers from one type to another. For these reasons, it is preferable in most cases that statistical analysis (and in this case, data mining) tasks be performed in a statistical analysis software package, rather than a DBMS.

This approach, however, introduces a compatibility issue, with regard to the data transferability: The data originating from the DBMS needs to somehow be transferred to the statistical analysis / data mining system. For this to be achieved, the data has to be exported by the DBMS in a way that is recognizable by the destination system – thus the DBMS needs to be able to export data in such a way – and then be imported in the destination system. This approach arises many issues that need to be taken into account, such as possible data alteration that may occur due to the export - import process, and, most importantly, the dependency of the whole process on the DBMS's capability to export the data in a format readable by the destination system.

In order to avoid having to go through all of these issues and to minimize the risk of a data integrity alternation, an alternative approach to the manner would be to fetch the data required directly from the DBMS into the destination system, without having to go through the export – import process at all. And thankfully, R provides all the tools needed for the job.

3.2 Required packages

In order to be able to connect to a database from within the R environment, extension packages need to be installed. There is a separate package for almost every DBMS that R supports, due to the fact that each DBMS requires a different communication interface implementation. Most of the packages, however, depend on a base package, the “DBI” package, which contains an interface definition for communication between R and DBMS’s – i.e. virtual / abstract classes, which need to be accordingly extended by the various DBMS communication package implementations.

The DBMSs that will be covered in this thesis are “MySQL” and “PostgreSQL”. The required package for R to communicate with a MySQL DBMS is the “RMySQL” package, while the package required for R to communicate with a PostgreSQL DBMS is the “RPostgreSQL” package. Both are available from the CRAN package repository, and instructions regarding their installation can be found in chapter 2.3.7 “Package installation in R”.

3.3 MySQL

3.3.1 A short history of MySQL

MySQL is (as of July 2013) the world's second most widely used open-source relational database management system (RDBMS) [20].

It was created by a Swedish company, MySQL AB, founded by David Axmark, Allan Larsson and Michael "Monty" Widenius. The first version of MySQL appeared on 23 May 1995. It was initially created for personal usage from mSQL based on the low-level language ISAM, which the creators considered too slow and inflexible. They created a new SQL interface, while keeping the same API as mSQL.

In January 2008, Sun Microsystems bought MySQL for $1 billion, and in April 2009, Oracle Corporation entered into an agreement to purchase Sun Microsystems, then owners of MySQL copyright and trademark.

In August 2012, TechCrunch's Alex Williams reported that Oracle was holding back MySQL Server test cases, a move that he concluded indicated that Oracle is attempting to kill the product. Since the final quarter of 2012, several Linux distributions and some important users (like Wikipedia and Google) started to replace MySQL with MariaDB [20].

3.3.2 Connecting R to MySQL

To connect from R to a MySQL DBMS, one should go through a procedure like the following:

- Install the “DBI” and “RMySQL” packages to R
- Load both packages
- Create a “MySQLDriver” object
- Create a “MySQLConnection” object

After installing the “DBI” and “RMySQL” packages, they need to be loaded:

```
> library(“RMySQL”)

Loading required package: DBI
```

If the above output is displayed, the packages have been successfully loaded into R.

Next, a “MySQLDriver” object needs to be created, which specifies the communication interface that is going to be utilized, via the “dbDriver()” function. In this case, the target system is a MySQL DBMS, so the driver to be instantiated should be the driver for communicating with a MySQL DBMS, which is called “MySQL”:

```
> drv <- dbDriver("MySQL")
> drv

<MySQLDriver:(1921)>
```

Once a driver object has been successfully created (i.e. an output like the above is displayed, and no errors come up), the system is ready to initiate a connection to the MySQL DBMS. For this, a “MySQLConnection” object needs to be created, using the “dbConnect()” function. The function parameters are: “host”, which specifies the database server (if not specified, it is assigned the default value of “localhost”), “dbname”, which defines the target database, “user”, which defines the database username, and “password”, which defines the user’s password:

```
> con <- dbConnect(drv,dbname="testdb",user="root",
+ password="",host=”localhost”)
> con

<MySQLConnection:(1921,2)>
```

In the case an output like the one above is displayed, the connection to the MySQL DBMS has succeeded, and R is now ready to send SQL queries to it and fetch data from it.

3.3.3 Issuing MySQL queries from R

To issue a MySQL query to the MySQL DBMS from R, a variety of functions are available by the "DBI" package. Some of the most basic functions are the following:

- dbWriteTable – Create a new remote table in the target database.
- dbRemoveTable() – Drop a remote table.dbListTables() – List remote tables, fields of a remote table, opened connections and pending statements in a connection.
- dbListFields () – List the fields of a remote table
- dbExistsTable() – Check whether a remote table in the database exists.
- dbSendQuery() – Submits and executes an arbitrary SQL statement on a specific connection. Also, clears (closes) a result set.
- dbGetQuery() - Submits and executes an arbitrary SQL statement on a specific connection, and fetches the result. Also, clears (closes) a result set.
- dbCommit() – Commit SQL transactions. (provided that the target system's MySQL version supports the transactions management feature).
- dbRollback() – Rollback SQL transactions (provided that the target system's MySQL version supports the transactions management feature).

The following examples demonstrate the basic usage of some of the above functions:

– Create a new remote table using "dbWriteTable()". Parameters: the database connection, the name of the table, and a data frame (or an object coercible to a data frame) which contains the table's rows.

```
> table.contents <-
+ data.frame(id=c(1,2,3),name=c("John","Mary","Jane"))
> dbWriteTable(con,name="mytable",value=table.contents)

[1] TRUE
```

– Drop the table:

```
> dbRemoveTable(con,"mytable")

[1] TRUE
```

– List the fields of the table:

```
> dbListFields(con,"mytable")

[1] "row.names" "id"        "name"
```

Though the fields specified upon the creation of the table were “id” and “name”, the table has been created with an extra field named “row_names”, which contains the row names of the table. Creating a “row_names” field is the default behavior of the “dbWriteTable()” function, and if such a field is not desirable, the “dbWriteTable()” function should be called with the parameter “row.names” set to FALSE, thus:

```
> dbRemoveTable(con,”mytable”)
> dbWriteTable(con,name="mytable",value=table.contents,
+ row.names=FALSE)

[1] TRUE
```

– Check if a remote table exists:

```
> dbExistsTable(con,”mytable”)

[1] TRUE
```

– Submit an SQL query:

```
> res <- dbSendQuery(con,”INSERT INTO mytable(id,name)
+ VALUES(2,’George’)”)
> res

<MySQLResult:(1921,2,59)>
```

The result of the “dbSendQuery()” function should always be stored in an object, in order to be able to use it to fetch the result records or close the result set.

– Submit an SQL query, and also fetch the result:

```
> dbGetQuery(con,"SELECT * FROM mytable")

  id name
1  1 John
2  2 Mary
3  3 Jane
```

The “dbGetQuery()” function sends a query to the DBMS and also fetches the result set back (as opposed to “dbSendQuery()”, which only sends the query to the DBMS). If the query is not a query that will produce a result set (for example, an INSERT or an UPDATE statement), then the returned result set will be NULL.

Contrary to sending information to a MySQL database, the procedure to fetch information from it implies the use of some of the following functions:

- dbReadTable() – Reads a remote table.
- fetch() – Fetches records from a result set.
- dbGetStatement() – Returns the statement that produced a result set.
- dbGetRowCount() – Returns the number of rows fetched from a result set.
- dbGetRowsAffected() – Returns the number of the affected rows of a statement.
- dbColumnInfo() – Returns the result set data types.
- dbHasCompleted() – Returns TRUE if are there more rows to fetch from a result set, false if there are not.

The following examples give an idea of the basic usage of some of the above functions:

– Read the newly created table's contents using "dbReadTable()":

```
> dbReadTable(con,"mytable")
```

```
  id name
1  1 John
2  2 Mary
3  3 Jane
```

– Get the statement that produced a result set:

```
> res <- dbSendQuery(con,"DELETE FROM mytable WHERE id=5")
> dbGetStatement(res)
```

```
[1] "DELETE FROM mytable WHERE id=5"
```

– Get the number of rows fetched:

```
> res <- dbSendQuery(con,"SELECT * FROM mytable")
> fetch(res,-1)
> dbGetRowCount(res)
```

```
[1] 3
```

3.3.4 Fetching information from a MySQL database to R

Given such a wide range of available database handling functions, what is left to discuss is how can the information fetched from a database be stored somewhere (for example, in an R object) so that it can be further manipulated by the statistical analysis / data mining system.

Most of the functions mentioned above return any fetched data from the DBMS as a data frame. Therefore, pretty much any information fetched from a DBMS can be stored in a data frame, and thus be further manipulatable by R.

Below is a small example demonstrating the procedure of connecting to a MySQL DBMS, fetching information from a database and storing it into an R data frame:

```
> library("RMySQL")
> drv <- dbDriver("MySQL")
> con <- dbConnect(drv,db="testdb",user="root",password="")
> res <- dbGetQuery(con,"SELECT * FROM mytable")
> class(res)

[1] "data.frame"

> res

  id name
1  1 John
2  2 Mary
3  3 Jane
```

Thus, the contents of the remote table "mytable" are now stored in the object "res".

3.4 PostgreSQL

PostgreSQL, often simply Postgres, is an open source object-relational database management system (ORDBMS) with an emphasis on extensibility and standards compliance. It is released under the PostgreSQL License, a free/open source software license. PostgreSQL is developed by the PostgreSQL Global Development Group. It implements the majority of the SQL:2011 standard, is ACID-compliant, is fully transactional (including all DDL statements), has extensible updateable views, data types, operators, index methods, functions, aggregates, procedural languages, and has a large number of extensions written by third parties. PostgreSQL is available for many platforms including Linux, FreeBSD, Solaris, Microsoft Windows and Mac OS X [21].

3.4.1 A short history of PostgreSQL

PostgreSQL evolved from the Ingres project at the University of California, Berkeley. In 1982, the project leader, Michael Stonebraker, left Berkeley to make a proprietary version of Ingres. He returned to Berkeley in 1985 and started a post-Ingres project to address the problems with contemporary database systems that had become increasingly clear during the early 1980s. The new project, POSTGRES, aimed to add the fewest features needed to completely support types.

In 1994, Berkeley graduate students Andrew Yu and Jolly Chen replaced the Ingres-based QUEL query language interpreter with one for the SQL query language, creating Postgres95. The code was released on the web.

In July 1996, Marc Fournier at Hub.Org Networking Services provided the first non-university development server for the open-source development effort. Along with Bruce Momjian and Vadim B. Mikheev, work began to stabilize the code inherited from Berkeley. The first open-source version was released on August 1, 1996.

In 1996, the project was renamed to PostgreSQL to reflect its support for SQL. The first PostgreSQL release formed version 6.0 in January 1997. Since then, the software has been maintained by a group of database developers and volunteers around the world, coordinating via the Internet.

The PostgreSQL project continues to make major releases (approximately annually) and minor "bugfix" releases, all available under the same license. Code comes from contributions from proprietary vendors, support companies, and open-source programmers at large [21].

3.4.2 Connecting R to PostgreSQL

The procedure of connecting to a PostgreSQL DBMS from R is identical to the procedure that should be followed to connect to a MySQL DBMS from R, described in chapter 3.3.2 “Connecting R to MySQL”, differing only in that the driver that needs to be instantiated for the database connection is the “RPostgreSQL” driver, instead of the “MySQL” one. Apart from that, the procedure is the same:

```
> drv <- dbDriver("PostgreSQL")
> con <- dbConnect(drv,db="testdb",user="root",password="",
+ host="localhost")
```

3.4.3 Issuing queries to a PostgreSQL DBMS from R

Since all of the R database querying functions are part of the “DBI” package, which both “RMySQL” and “RPostgreSQL” packages depend on, the functions used and the way they are meant to be used are identical, either querying a MySQL DBMS, or querying a PostgreSQL DBMS. Therefore, the procedure of issuing queries to a PostgreSQL DBMS from R is identical to that of issuing queries to a MySQL DBMS from R, described in chapter 3.3.3 “Issuing MySQL queries from R”.

3.4.4 Fetching information from a PostgreSQL database to R

The R functions used for fetching data from a DBMS are also part of the “DBI” package, on which, as mentioned before, the packages “RMySQL” and “RPostgreSQL” depend. Consequently, fetching information from a PostgreSQL DBMS to R is identical to fetching information from a MySQL DBMS to R, described in chapter 3.3.4 “Fetching information from a MySQL database to R”.

Below is a small example demonstrating the procedure of connecting to a PostgreSQL DBMS, fetching information from a database and storing it into an R data frame:

```
> library(“RPostgreSQL”)
> drv <- dbDriver("PostgreSQL ")
> con <- dbConnect(drv,db="testdb",user="postgres",
+ password="")
> res <- dbGetQuery(con,”SELECT * FROM mytable”)
> class(res)

[1] "data.frame"

> res

  id name
1  1 John
2  2 Mary
3  3 Jane
```

3.4.5 Conclusion

It can easily be concluded that R is perfectly capable of sufficiently handling the communication with a DBMS, as it provides functions for all of the major operations of a DBMS – using only two extension packages and three commands, R is set up and ready to communicate with a DBMS. With this information in mind, the system on which the case studies for the sakes of this thesis were conducted will be set up, a process described in the next chapter.

4. R installation to the dbTech.net Virtual Machine

4.1 Introduction and purpose

In this chapter, the setting up of the system on which the case studies for the sakes of this thesis were conducted will be described. The software that will finally be present in the system is:

- Debian Linux, version 6.0.7 (Squeeze), Kernel Version 2.6.32-5-686 (installed beforehand)
- PostgreSQL Server, version 8.4
- pgAdmin, version 1.10.5 (Aug 1 2010)
- MySQL, version 5.6.12 (installed beforehand)
- R, version 3.0.2 (2013-09-25)
- R package “arules” , version 1.1-0
- R package “arulesViz”, version 0.1-7
- R package “scatterplot3d”, version 0.3-34
- R package “vcd”, version 1.3-1
- R package “colorspace”, version 1.2-4
- R package “igraph”, version 0.6.6
- R package “DBI”, version 0.2-7
- R package “RPostgreSQL”, version 0.4
- R package “gclus”, version 1.3.1
- R package “seriation”, version 1.0-11
- R package “TSP”, version 1.0-8
- R package “rjava”, version 0.9-5
- R package “RWeka”, version 0.4-21
- R package “FSelector”, version 0.19
- R package “partykit”, version 0.1-6

All system packages will be installed from the default aptitude repositories, where possible. All R packages will be installed from the CRAN package repository, where possible. To perform the following actions, root access to the system is required.

In the case that a package installation from the default aptitude repositories or the CRAN package repository fails, official mirrors will be used alternatively.

4.2 The DBTechNet Virtual Machine

The case studies were conducted in a Virtual Machine environment using a Virtual Machine image provided by DBTechNet. The allocated memory to it was 1536MB,

the processor core allocated to it belonged to an Intel Core 2 Duo Q6600 processor, clocked at 2,4GHz, and the software used to operate the virtual machine was Oracle VM VirtualBox, version 4.2.18.

DBTechNet is an initiative of European higher education institutions and IT-companies to set up a transnational collaboration scheme of higher level educational establishments, IT enterprises and vocational training centers that will collaborate in order to achieve a three-fold goal, namely:

- Developing efficient Internet based tools (like a Web-based IT terminology encyclopedia, universal database access terminal, etc.) which will organize worldwide access to database technology resources and educational/training material and references
- Design and develop virtual workshop type course modules on selected database topics that will address the wide spectrum of new trends, backed by online support from a network of educational and IT professional experts
- Promote entrepreneurship by developing a business plan of operation, which will make it possible for the collaboration scheme in question to evolve into a self-sustained consortium that will function within the new emerging reality of education and vocational training in today's information and communication technology driven society. [1]

4.3 Necessary pre-software installation actions

After successfully importing and booting the virtual machine, the first thing that needs to be done is an aptitude update, in order to make sure that the package information is up to date, and thus the latest versions of each package will be downloaded.

To initiate an aptitude update, the "apt-get update" command needs to be entered in a system terminal. In Debian Linux version 6.0.7, a terminal window can be opened by using the operating system's top menus and navigating to the "Applications" menu – "Accessories" – "Terminal".

Once a terminal window is open, root access to the system is required to be gained, which is achieved using the "su" command and entering the root password.

Once root access to the system has been obtained, prior to updating the aptitude repositories a CRAN repository needs to be added in the repositories list file. For the sakes of this thesis, the Greek CRAN repository was used. The adding of the repository to the repositories list file is achieved thus:

```
# echo 'deb http://cran.cc.uoc.gr/bin/linux/debian squeeze-
cran3/' >> /etc/apt/sources.list
```

After successfully inserting the repository to the repositories list file, the aptitude update is in order:

```
# apt-get update
```

The system will now scan all of the mirrors specified in "/etc/apt/sources.list". The moment the scan is complete, the system's aptitude package related information is up to date.

4.4 R setup

Obviously, the software installation process will begin with the installation of the R language. R can be installed from the aptitude repositories:

```
# apt-get install r-base r-base-dev
```

The total size of the r-base package and its dependencies (i.e. the packages required to be installed in order for "r-base" to work correctly) is about 85 megabytes, thus they will take some time to download.

To ensure that R was successfully installed, it can be executed by using the "R" command in the terminal window:

```
# R
```

4.5 Installation of PostgreSQL

After successfully installing R, it is time for PostgreSQL to be installed. The packages that need to be installed are: "postgresql", "postgresql-client", "pgadmin3", and "postgresql-server-dev-all".

```
# apt-get install postgresql postgresql-client pgadmin3
postgresql-server-dev-all postgresql-server-dev-8.4
```

4.6 Installation of required packages

After a successful installation of R and PostgreSQL, the packages required for the conduction of the case studies of this thesis need to be installed. All of the necessary packages are previously mentioned in chapter 4.1 "Introduction and

purpose". The preferred source to install a package is an official CRAN mirror; however, in many cases the package cannot be installed from a mirror, thus an alternative source is used.

Installation of the "RPostgreSQL" package (includes the installation of the "DBI" package):

```
# R
> install.packages("RPostgreSQL")
```

Installation of the "arulesViz" package (includes the installation of the "arules", "igraph", "TSP", "scatterplot3d", "vcd", "colorspace", "gclus" and "seriation" packages):

```
> install.packages("arulesViz")
```

Installation of the "RWeka" package (includes the installation of the "rjava" package):

```
# R CMD javareconf
# R
> install.packages("RWeka")
```

Installation of the "party" package (includes the installation of the "coin", "zoo", "sandwich" and "strucchange" packages):

```
> install.packages("party")
```

Installation of the "partykit" package:

```
> install.packages("partykit")
```

Installation of the "FSelector" package:

```
# R CMD javareconf       /*only if not previously performed*/
# R
> install.packages("FSelector")
```

4.7 Conclusion

After successfully installing the software and the required packages, the system is set up and ready. To verify the successful installation of every package, each

package must be loaded into R using the “library()” function for each one, as described in chapter 2.3.8 “Verification of a package installation in R”.

5. Case Studies

This chapter is devoted to the case studies that were conducted for the sakes of this thesis. Two case studies will be demonstrated and detailed: The first case study demonstrates how association rules are mined in R from a supermarket transactions dataset, how these rules are visualized and evaluated, and how they can be stored in a DBMS database for permanent storage and further processing. The algorithm used to extract the association rules is the Apriori algorithm. Furthermore, it demonstrates the creation of a product recommender system to the customers of a supermarket. The second case study is conducted on a dataset which contains information about the passengers of the Titanic, the British passenger liner that sank in the North Atlantic Ocean in 1912, and demonstrates the use of a classification method in R in order to classify the passengers according to their survival or not. The classification method used is the construction of a Decision Tree using the C4.5 algorithm.

5.1 Mining association rules from supermarket transactions

5.1.1 Introduction

The first case study demonstrates how association rules are mined from a dataset using R and the Apriori algorithm. The dataset contains 247535 records consisting of 2037 items contained in 33701 transactions of a supermarket, and is the dataset used in IBM's data mining tutorial "Mining your business in retail with IBM DB2 Intelligent Miner"[3]. Each row of the dataset refers to one item. The objective of the case study was to extract useful information about the items by finding significant relations among items so that the supermarket can rearrange its marketing strategies accordingly.

Following is a short insight into the steps taken in order to carry out the association rule mining:

- Loading the required packages in R
- Pre-processing - reading the transactions into R
- Pre-processing - extracting the transactions information that will be used in the association rules mining process
- Performing the association rules mining
- Visualizing and evaluating the results
- Storing the results in a database, for permanent storage and further processing

5.1.2 Loading the required packages into R

Beginning the procedure, the first thing required to do is to load all of the packages that are going to be used. All of those packages are dependency packages of the "arulesViz" package, thus when "arulezViz" is loaded, every package required by it is loaded as well. Therefore, the action needed to be taken is to load the "arulesViz" package:

```
> library("arulesViz")

Loading required package: arules
Loading required package: Matrix
Loading required package: lattice

Attaching package: 'arules'

The following object is masked from 'package:base':

    %in%, write

Loading required package: scatterplot3d
Loading required package: vcd
Loading required package: grid
Loading required package: seriation
Loading required package: cluster
Loading required package: TSP
Loading required package: gclus
Loading required package: colorspace

Attaching package: 'seriation'

The following object is masked from 'package:lattice':

    panel.lines

Loading required package: igraph

Attaching package: 'igraph'

The following object is masked from 'package:gclus':

    diameter


Attaching package: 'arulesViz'

The following object is masked from 'package:base':

    abbreviate
```

If an output like the above is displayed and no errors occur, the packages have been successfully loaded into R.

5.1.3 Pre-processing - reading the transactions into R

After successfully loading the required packages, the transaction data needs to be read by R in order to be able to process it further. The transactions dataset is located in a csv file named “pos_data.csv”. In order to read it, two possible functions can be used – the “read.csv()” function and the “read.table()” function. Though “read.csv()” seems the more appropriate choice in this case, the “read.table()” is actually a better option, because it offers much more control over the data import than “read.csv()” does.

To read the transactions, the following command needs to be issued:

```
> transactions.table <-  read.table("pos_data.csv",
+ header=F,sep="|",quote="",dec=".",col.names=c("customer_id",
+ "trans_id","item_id","trans_date", "trans_time","quantity","price",
+ "promotion"), colClasses=c("numeric","numeric","numeric","character",
+ "numeric", "numeric","numeric","numeric"),strip.white=T,
+ blank.lines.skip=T,comment.char="",allowEscapes=F)
```

Parameters explanation:

- ‘header=F’ – instructs the function that it should not regard the first line of the dataset as a line containing the column names, as this is not the case in this dataset.
- ‘sep="|"’ – instructs the function that the character used to separate the entries in the dataset is not the comma character (“,”) but the “|” (“vertical bar”) character instead.
- ‘quote=""’ – instructs the function that quoting of the read data is not desired.
- ‘dec="."’ – instructs the function that the character used for decimal points is the dot (“.”) character.
- ‘col.names=c(…)’ – used to give a name to each column of the dataset.
- ‘colClasses = c(…)’ – used to define the class as which each column will be read.
- ‘strip.white=T’ – instructs the function to strip any leading and trailing white space from unquoted character entries.
- ‘blank.lines.skip=T’ – instructs the function to not regard blank lines as records.
- ‘comment.char=""’ – instructs the function that no character should be regarder as reserved for commenting.
- ‘allowEscapes=F’ – instructs the function that no character sequence should be regarded as escape characters (for example, character sequences like “\n” or “\t”).

5.1.4 Pre-processing - extracting the transactions information that will be used in the association rules mining process

Once successfully reading the transaction data, the next step to take is to extract only the information needed for the association rules mining process. As mentioned in the case study description, each row of the dataset contains information about one item, and not about a complete transaction. Thus, every record consists of information about the item – its ID, its price, its being a promotional item or not, the quantity in which it was purchased, the date and time it was purchased, the ID of the customer that purchased it, and the ID of the transaction it belongs to. While all of this information is certainly useful to know, nevertheless it is not needed in its entirety in the association rules mining process – the only part of a transaction that is relevant for an association rules mining process is the item ID and the transaction ID to which this item belongs to.

Consequently, the item and transaction IDs need to be extracted from the dataset, and placed into a separate object, which will then be used by the association rules mining algorithm:

```
> transactions <- transactions.table[c("trans_id","item_id")]
```

In this case study, the items themselves are not going to be used in the association rules mining process, but rather the category they belong to is. The information about the category that each item belongs to is held in the "article_categories.csv" file, while the category names are held into the "articles.csv" file. Thus, the aim is to create a new relational table which will hold the information about the categories of the items purchased in each transaction. This table is named "items_to_categories". This can be achieved using SQL: Firstly, the tables "articles" and "article_categories" need to be created, which will hold the information of the "articles.csv" and "article_categories.csv" files respectively, so that the "items_to_categories" table can be created afterwards.

```
> articles <- read.table("articles.csv",header=F,sep="|",quote="",
+ dec=".",col.names=c("item_id","cat_id"),colClasses=c("numeric",
+ "numeric"),strip.white=T,blank.lines.skip=T,comment.char="",
+ allowEscapes=F)
> dbWriteTable(con.postgresql,name="articles",value=articles)
> article_categories <-
read.table("article_categories.csv",header=F,sep="|",quote="",
+ dec=".",col.names=c("cat_id","cat_desc"),colClasses=c("numeric",
+ "character"),strip.white=T,blank.lines.skip=T,comment.char="",
+ allowEscapes=F)
> dbWriteTable(con.postgresql,name="article_categories",
+ value=article_categories)
```

After creating the two tables, the "items_to_categories" table can be created. To create this table, it is required that all of the transactions are scanned, the categories of the items each transaction contains are determined, and a record is

placed in the new table containing the transaction ID and the items' categories names:

```
> con.postgresql <- dbConnect(dbDriver("PostgreSQL"),dbname="postgres",
+ user="postgres",password="my$q7",host="localhost")
> for (i in 1:length(transactions$item_id)) {
+     cat_desc <- dbGetQuery(con2,paste("SELECT cat_desc AS cd FROM
+     article_categories AC INNER JOIN articles A ON A.cat_id =
+     AC.cat_id WHERE A.item_id = '",transactions$item_id[i],"'",
+     sep=""))
+     dbGetQuery(con.postgresql,paste("INSERT INTO
+     items_to_categories(trans_id,item_id)
+     VALUES('",transactions$trans_id[i],"','",cat_desc$cd,"')"))
+ }
```

After successfully creating the "items_to_categories" table, the transactions are ready to be read from it by R:

```
> transactions <- dbReadTable(con.postgresql,"items_to_categories")
> dbDisconnect(con.postgresql)
```

After reading the transactions from the newly created table, the new dataset (which is a "data.frame" class object) needs to be converted into a "transactions" class object. In order to achieve this, it needs to be written in a file which will then be read by the "read.transactions()" function, which reads a dataset containing transactions and creates a "transactions" object out of it:

```
> #write the transactions to a file:
> write.table(transactions,file="trans_file",row.names=F,quote=F)
```

Parameters explanation:

- 'transactions' – the source dataset.
- 'file="trans_file"' – the destination file.
- 'row.names=F' – instructs the function to not append an extra column to the file, containing the row names of the dataset.
- 'quote=F' – instructs the function to not quote the entries.

```
> #create a "transactions" object from the file:
> transactions.final <- read.transactions("trans_file",format="single",
+ cols=c("trans_id","item_id"),rm.duplicates=T)
```

Parameters explanation:

- '"trans_file"' – the file to read the transactions from.
- 'format="single"' – instructs the function that the transactions are stored in the dataset using the "single" format, which indicates that each row contains

a single item and the transaction to which it belongs, and not a complete transaction.

- 'cols=c("trans_id","item_id")' – shows the algorithm which columns of the dataset are to be regarded as the transaction ID column and the item ID column, respectively.
- 'rm.duplicates=T' – instructs the algorithm to remove duplicate entries.

5.1.5 Performing the association rules mining

After having read the transactions in a "transactions" class object, the data pre-processing phase is completed, and the transactions are ready to be mined for association rules. The rule are mined using the "apriori()" function. The acceptable rules need to have a support measure greater than 0.02, a confidence measure greater than 0.02 and a minimum item length of 2:

```
> rules <- apriori(transactions.final,parameter=list(supp=0.02,
+ conf=0.02,minlen=2))

parameter specification:
 confidence minval smax arem  aval originalSupport support minlen maxlen
target   ext
        0.3    0.1    1 none FALSE            TRUE    0.1       2     10
rules FALSE

algorithmic control:
 filter tree heap memopt load sort verbose
    0.1 TRUE TRUE  FALSE TRUE    2    TRUE

apriori - find association rules with the apriori algorithm
version 4.21 (2004.05.09)        (c) 1996-2004   Christian Borgelt
set item appearances ...[0 item(s)] done [0.00s].
set transactions ...[71 item(s), 33701 transaction(s)] done [0.01s].
sorting and recoding items ... [13 item(s)] done [0.00s].
creating transaction tree ... done [0.01s].
checking subsets of size 1 2 3 done [0.00s].
writing ... [26 rule(s)] done [0.00s].
creating S4 object  ... done [0.00s].
```

5.1.6 Visualizing and evaluating the results

Thus, the association rules have been mined. Observing the output of the "apriori()" function, information about its progress can be gained – for example, the total different items contained in the transactions were 71, the transactions added up to a total of 33701, and the rules mined were 26.

To see the mined rules, the "inspect()" function is used:

```
> inspect(rules)
```

```
   lhs                                rhs
support   confidence     lift
1  {PROCESSED FOOD(EXC. SOUPS)} => {MILK,CHEESE,EGGS}
0.1105902  0.7965377 2.176785
2  {MILK,CHEESE,EGGS}           => {PROCESSED FOOD(EXC. SOUPS)}
0.1105902  0.3022219 2.176785
3  {COFFEE,TEA,CACAO,TOBACCO}   => {MILK,CHEESE,EGGS}
0.1058722  0.6001682 1.640145
4  {BACKED GOODS}               => {FRESH MEAT,SAUSAGES,FISH}
0.1013620  0.5650017 2.284752
5  {FRESH MEAT,SAUSAGES,FISH}   => {BACKED GOODS}
0.1013620  0.4098872 2.284752
6  {BACKED GOODS}               => {MILK,CHEESE,EGGS}
0.1368505  0.7628184 2.084637
7  {MILK,CHEESE,EGGS}           => {BACKED GOODS}
0.1368505  0.3739864 2.084637
8  {BEER,ALCH. FREE DRINKS}     => {CONFECTIONERY}
0.1112430  0.4264589 1.379279
9  {CONFECTIONERY}              => {BEER,ALCH. FREE DRINKS}
0.1112430  0.3597889 1.379279
10 {BEER,ALCH. FREE DRINKS}     => {MILK,CHEESE,EGGS}
0.1261684  0.4836765 1.321795
11 {MILK,CHEESE,EGGS}           => {BEER,ALCH. FREE DRINKS}
0.1261684  0.3447940 1.321795
12 {SOUPS,SPICES,SPREAD}        => {FRESH MEAT,SAUSAGES,FISH}
0.1049227  0.5638654 2.280157
13 {FRESH MEAT,SAUSAGES,FISH}   => {SOUPS,SPICES,SPREAD}
0.1049227  0.4242861 2.280157
14 {SOUPS,SPICES,SPREAD}        => {CONFECTIONERY}
0.1016884  0.5464838 1.767471
15 {CONFECTIONERY}              => {SOUPS,SPICES,SPREAD}
0.1016884  0.3288868 1.767471
16 {SOUPS,SPICES,SPREAD}        => {MILK,CHEESE,EGGS}
0.1457227  0.7831287 2.140141
17 {MILK,CHEESE,EGGS}           => {SOUPS,SPICES,SPREAD}
0.1457227  0.3982322 2.140141
18 {FRESH MEAT,SAUSAGES,FISH}   => {CONFECTIONERY}
0.1230824  0.4977202 1.609757
19 {CONFECTIONERY}              => {FRESH MEAT,SAUSAGES,FISH}
0.1230824  0.3980806 1.609757
20 {FRESH MEAT,SAUSAGES,FISH}   => {MILK,CHEESE,EGGS}
0.1862556  0.7531797 2.058296
21 {MILK,CHEESE,EGGS}           => {FRESH MEAT,SAUSAGES,FISH}
0.1862556  0.5090010 2.058296
22 {CONFECTIONERY}              => {MILK,CHEESE,EGGS}
0.1815376  0.5871401 1.604542
23 {MILK,CHEESE,EGGS}           => {CONFECTIONERY}
0.1815376  0.4961077 1.604542
24 {CONFECTIONERY,
    FRESH MEAT,SAUSAGES,FISH}   => {MILK,CHEESE,EGGS}
0.1042106  0.8466731 2.313796
25 {FRESH MEAT,SAUSAGES,FISH,
    MILK,CHEESE,EGGS}           => {CONFECTIONERY}
0.1042106  0.5595029 1.809579
26 {CONFECTIONERY,
    MILK,CHEESE,EGGS}           => {FRESH MEAT,SAUSAGES,FISH}
0.1042106  0.5740438 2.321316
```

At first glance, it is perceivable that all of the rules have low support values, which leads to the conclusion that this transactions dataset does not include any highly significant relationship between any of its contained items. Nevertheless, the confidence measure of most of the rules ranges from ~0.3 to ~0.84, which means that the conditional probability of a transaction containing the items in the body of the rule to also contain the item on the head of the rule ranges from ~30% to ~84%, which is not at all bad. Finally, the lift measures of the rules range from ~1.3 to ~2.32, thus indicating little importance in the mined rules. One of the most important rules of this mining is rule number 24 {CONFECTIONERY, FRESH MEAT,SAUSAGES,FISH} => {MILK,CHEESE,EGGS}), because of its relatively high confidence measure compared to the rest of the rules, which means that the conditional possibility of a customer buying confectionary and meat/fish products to also buy milk/cheese/eggs kind of products is ~84%. Other important rules are rule number 20 ({FRESH MEAT,SAUSAGES,FISH} => {MILK,CHEESE,EGGS}), because of its comparatively high lift value (~2.06), its high support measure compared to the rest of the rules (~0.19) and its high confidence value (~0.75), or rule number 6 ({BACKED GOODS} => {MILK,CHEESE,EGGS}), with a support of ~0.14, a confidence of ~0.76 and a lift of ~2.08, or rule number 16 ({SOUPS,SPICES,SPREAD} => {MILK,CHEESE,EGGS}), with a support of ~0.15, a confidence of ~0.78 and a lift of ~2.14, or rule number 1 ({PROCESSED FOOD(EXC. SOUPS)} => {MILK,CHEESE,EGGS}), with a support value of ~0.11, a relatively high confidence value of ~0.80, and a relatively high lift value of ~2.18.

These results can become clearer to the eye by using a visualization method to display them. For example:

```
> plot(rules,measure=c("confidence","lift"),shading="support")
```

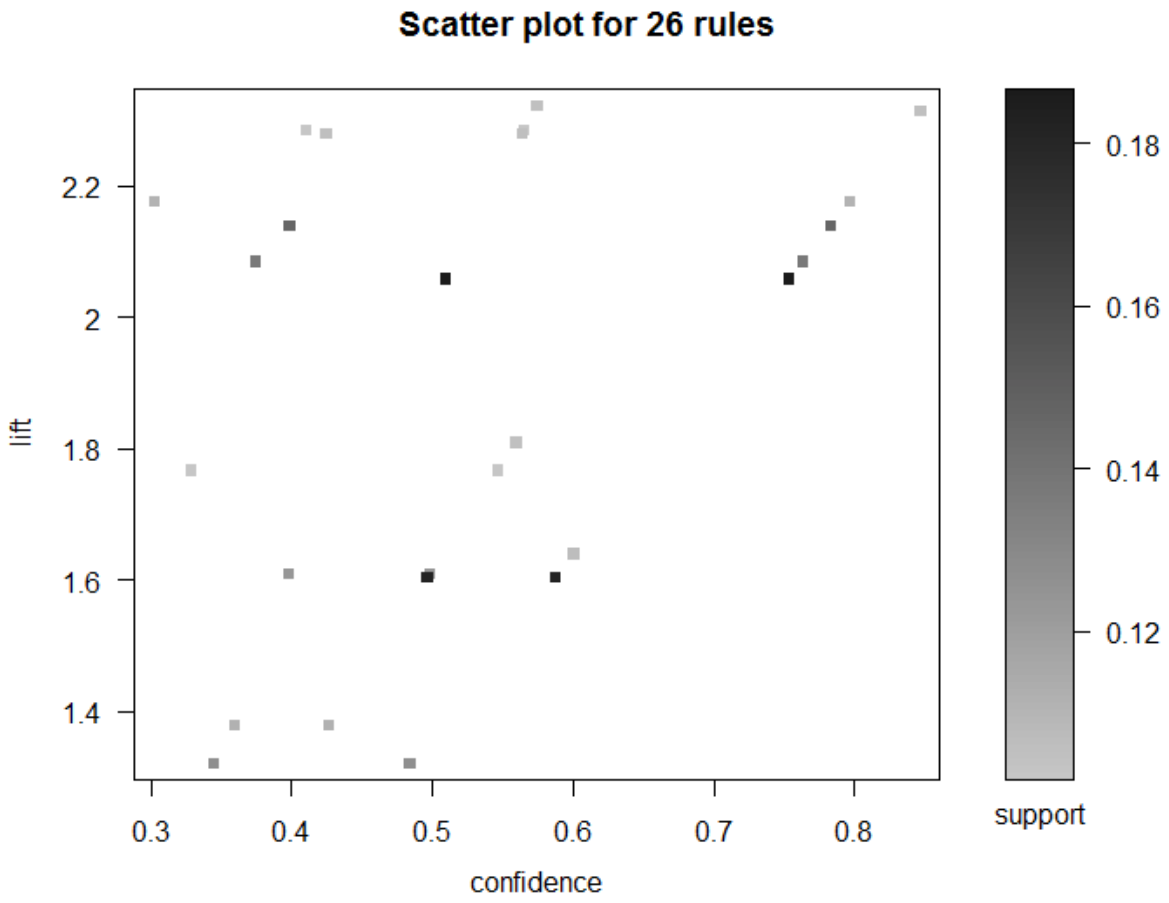

Figure 5.1.6-1: Visual representation of the rules via scatter plot

In this graph, it is clear that the rule located on the top right of the graph is rule number 24 {CONFECTIONERY, FRESH MEAT,SAUSAGES,FISH} => {MILK,CHEESE,EGGS}), because of its comparatively high lift and confidence values but its poor support value. Likewise, the other 4 rules located to the top right region and underneath rule number 24 are some of the most important ones, among which are rule number 20 ({FRESH MEAT,SAUSAGES,FISH} => {MILK,CHEESE,EGGS}), rule number 6 ({BACKED GOODS} => {MILK,CHEESE,EGGS}), rule number 16 ({SOUPS,SPICES,SPREAD} => {MILK,CHEESE,EGGS}) and rule number 1 ({PROCESSED FOOD(EXC. SOUPS)} => {MILK,CHEESE,EGGS}), all of which were mentioned previously.

5.1.7 Storing the results in a database, for permanent storage and further processing

After evaluating the rules, it may be desired that they are permanently stored somewhere for further processing or just for their availability in case of future

reference. A database in a DBMS can be considered such a place, due to the DBMSs emphasis on many aspects regarding data integrity, security and availability. In this case study, a PostgreSQL DBMS database named “rulesdb” is used, hosted locally and accessed by the root user using the password “password”.

To store the rules in a database, the first thing needed to be made is a connection to the database:

```
> library("RPostgreSQL")  #load the appropriate communication package
> con.postgresql <- dbConnect(dbDriver("PostgreSQL"),dbname="rulesdb",
+ user="root",password="password",host="localhost")
```

After successfully connecting to the database, the data can be inserted to it. In this case study, the rules are inserted to a table named “rules”, created during insertion time with the use of the “dbWriteTable()” function:

```
> dbWriteTable(con.postgresql,name=rules.tablename,
+ value=as(rules,"data.frame"))
```

Parameters explanation:

- ‘con.postgresql’ – the database connection
- ‘name=”rules”’ – the name of the table to insert the data into
- ‘value=as(rules,”data.frame”)’ – the data source to be inserted to the table, “on the fly” converted from a “transactions” class object to a “data.frame” class object

To make sure that the data was inserted in the database successfully, the table content is printed using the “dbReadTable()” function:

```
> dbReadTable(con.postgresql,”rules”)
```

```
                                                                rules
support confidence     lift
1              {PROCESSED FOOD(EXC. SOUPS)} => {MILK,CHEESE,EGGS}
0.1105902  0.7965377 2.176785
2              {MILK,CHEESE,EGGS} => {PROCESSED FOOD(EXC. SOUPS)}
0.1105902  0.3022219 2.176785
3                {COFFEE,TEA,CACAO,TOBACCO} => {MILK,CHEESE,EGGS}
0.1058722  0.6001682 1.640145
4                    {BACKED GOODS} => {FRESH MEAT,SAUSAGES,FISH}
0.1013620  0.5650017 2.284752
5                    {FRESH MEAT,SAUSAGES,FISH} => {BACKED GOODS}
0.1013620  0.4098872 2.284752
6                           {BACKED GOODS} => {MILK,CHEESE,EGGS}
0.1368505  0.7628184 2.084637
7                           {MILK,CHEESE,EGGS} => {BACKED GOODS}
0.1368505  0.3739864 2.084637
8                    {BEER,ALCH. FREE DRINKS} => {CONFECTIONERY}
0.1112430  0.4264589 1.379279
9                    {CONFECTIONERY} => {BEER,ALCH. FREE DRINKS}
0.1112430  0.3597889 1.379279
```

```
10                {BEER,ALCH. FREE DRINKS} => {MILK,CHEESE,EGGS}
0.1261684  0.4836765 1.321795
11                {MILK,CHEESE,EGGS} => {BEER,ALCH. FREE DRINKS}
0.1261684  0.3447940 1.321795
12             {SOUPS,SPICES,SPREAD} => {FRESH MEAT,SAUSAGES,FISH}
0.1049227  0.5638654 2.280157
13             {FRESH MEAT,SAUSAGES,FISH} => {SOUPS,SPICES,SPREAD}
0.1049227  0.4242861 2.280157
14                       {SOUPS,SPICES,SPREAD} => {CONFECTIONERY}
0.1016884  0.5464838 1.767471
15                       {CONFECTIONERY} => {SOUPS,SPICES,SPREAD}
0.1016884  0.3288868 1.767471
16                    {SOUPS,SPICES,SPREAD} => {MILK,CHEESE,EGGS}
0.1457227  0.7831287 2.140141
17                    {MILK,CHEESE,EGGS} => {SOUPS,SPICES,SPREAD}
0.1457227  0.3982322 2.140141
18                  {FRESH MEAT,SAUSAGES,FISH} => {CONFECTIONERY}
0.1230824  0.4977202 1.609757
19                  {CONFECTIONERY} => {FRESH MEAT,SAUSAGES,FISH}
0.1230824  0.3980806 1.609757
20                {FRESH MEAT,SAUSAGES,FISH} => {MILK,CHEESE,EGGS}
0.1862556  0.7531797 2.058296
21                {MILK,CHEESE,EGGS} => {FRESH MEAT,SAUSAGES,FISH}
0.1862556  0.5090010 2.058296
22                           {CONFECTIONERY} => {MILK,CHEESE,EGGS}
0.1815376  0.5871401 1.604542
23                           {MILK,CHEESE,EGGS} => {CONFECTIONERY}
0.1815376  0.4961077 1.604542
24 {CONFECTIONERY,FRESH MEAT,SAUSAGES,FISH} => {MILK,CHEESE,EGGS}
0.1042106  0.8466731 2.313796
25 {FRESH MEAT,SAUSAGES,FISH,MILK,CHEESE,EGGS} => {CONFECTIONERY}
0.1042106  0.5595029 1.809579
26 {CONFECTIONERY,MILK,CHEESE,EGGS} => {FRESH MEAT,SAUSAGES,FISH}
0.1042106  0.5740438 2.321316
```

Therefore, the rules were successfully inserted into the database, into a newly created table called “rules” and consisting of 4 fields: “rules”, “support”, “confidence” and “lift”.

5.1.8 Creating a recommender system

After mining the association rules from a supermarket transactions dataset, a wise marketing strategy would be for the supermarket to use the information gained from the association rules mining process to be able to recommend items to customers depending on what they are buying, the moment they are buying it. To achieve this, the supermarket needs to build what is called a “recommender system” – a system that recommends items to a customer depending on what they are buying at the moment. The recommendations are, of course, based on the supermarket’s records of other customers’ past transactions. Using this system, the supermarket will be able to know which items a customer is more likely to purchase along with those they already intend to.

For the sakes of this case study, an IBM's tutorial to guide application designers and application developers through the process of gaining insight from data mining analysis to applying it in end user applications was used, titled "Mining your Business in Retail with IBM DB2 Intelligent Miner" [3]. In this tutorial, IBM presents how to apply data mining techniques using IBM DB2 Intelligent Miner Modeling to automatically generate real-time product recommendations for customers in a possible e-commerce shop environment. The tutorial is carried out using IBM DB2 Intelligent Miner. This case study demonstrates how R can be used instead, in order to mine association rules from a supermarket transactions dataset and use them to create a recommender system – a system which generates real-time product recommendations for its customers – identical to the one created in IBM's tutorial.

In order to create a recommender system, the supermarket first needs to know the associations that exist among itemsets that customers purchase. In other words, the supermarket needs *the association rules* for the transactions conducted by its customers. After having mined the association rules, the recommended items for each transaction are going to be the items contained in the heads of the itemsets whose body consists of the items of the transaction in question. Clarifying the matter, if a customer is purchasing items x and y, and association rules of the forms "{x,y} => {z}" OR "{x} => {z}" OR "{y} => {z}" exist, and have sufficient association rule metrics, then item z is a prudent recommendation regarding this transaction.

For the sakes of this case study, the transaction(s) for which recommendations are going to be produced are contained in a database table named "new_transactions", and the transaction(s) data is contained in a data frame named "new_transactions". Three transactions are going to be processed: The first transaction consists of items "DIET FOOD", "MILK/CHEESE/EGGS" and "FRESH FRUITS/VEGETABLES", the second transaction consists of items "BACKED GOODS" and "COFFEE/TEA/CACAO/TOBACCO", and the third transaction consists of items "FROZEN FOOD/ICE CREAM" and "SOAP/BODY CARE PROD.". The "new_transactions" table and data frame are created as follows:

```
# create the transactions table that contains the transaction(s) for
+ which recommendations are going to be produced:
> new_transactions <- data.frame(transid=c(0),itemid=c(0))
> new_transactions <- new_transactions[-1,]
> for(i in 1:length(trans)) {
>     new_trans_id <- dbGetQuery(con.postgresql,"SELECT max(trans_id) AS
+     transid FROM transactions")[1] + i
>     new_transactions <- rbind(new_transactions,data.frame(
+     transid=c(new_trans_id[1]),itemid=trans[[i]],row.names=NULL))
> }
> if(dbExistsTable(con.postgresql,"new_transactions"))
>     dbRemoveTable(con.postgresql,"new_transactions")
> dbWriteTable(con.postgresql,name="new_transactions",
+ value=new_transactions)
```

```
> dbGetQuery(con.postgresql,'ALTER TABLE new_transactions DROP COLUMN
+ "row.names"')
```

In order to proceed with the recommender system creation, the rules' bodies and heads need to be able to be scanned separately. Right now the rules exist in a "{body} => {head}" form, and are contained in a single column in a database table, therefore the bodies and heads of each rule need to be separated from each other. Using the database table in which the association rules were stored previously (assuming it is called "rules"), five more columns need to be added, which will contain each rule's body and head, plus an ID number for each body, the item ID that each head consists of, and the rule length:

```
> dbGetQuery(con.postgresql,paste("ALTER TABLE ",rules.tablename," ADD
COLUMN bodytext VARCHAR(2500)",sep=""))
> dbGetQuery(con.postgresql,paste("ALTER TABLE ",rules.tablename," ADD
COLUMN headname VARCHAR(2500)",sep=""))
> dbGetQuery(con.postgresql,paste("ALTER TABLE ",rules.tablename," ADD
COLUMN bodyID INTEGER",sep=""))
> dbGetQuery(con.postgresql,paste("ALTER TABLE ",rules.tablename," ADD
COLUMN head INTEGER",sep=""))
> dbGetQuery(con.postgresql,paste("ALTER TABLE ",rules.tablename," ADD
COLUMN length INTEGER",sep=""))
```

After altering the rules table appropriately, the splitting of the heads and bodies of the rules needs to take place:

```
# split the head and body:
> rules.column <- dbGetQuery(con.postgresql,paste("SELECT rules FROM ",
+ rules.tablename,sep=""))
> rules.separated <- do.call('rbind',strsplit(rules.column[,1],"=>",
+ fixed=T))
> rules.bodies <- rules.separated[,1]
> rules.heads <- rules.separated[,2]
> for (i in 1:length(rules.bodies)) {
>     dbGetQuery(con.postgresql,paste("UPDATE ",rules.tablename,"
+     SET headname='",rules.heads[i],"',bodytext='",rules.bodies[i],"'
+     WHERE rules='",rules.bodies[i],"=>",rules.heads[i],"'",sep=""))
> }
```

After splitting the heads and bodies, bodies need to be given an ID number which shall be placed in the "bodyid" column, and each head's item needs to be determined – and placed in the 'head' column:

```
# Give IDs to the bodies, get the IDs of the items in the heads:
> cat("Giving IDs to the heads and bodies... ")
> distinct.bodies <- dbGetQuery(con.postgresql,paste("SELECT DISTINCT
+ bodytext FROM ",rules.tablename,sep=""))
> distinct.heads <- dbGetQuery(con.postgresql,paste("SELECT DISTINCT
+ headname FROM ",rules.tablename,sep=""))
> distinct.bodies$body_id <- c(1:length(distinct.bodies$bodytext))
> for (i in 1:length(distinct.heads$headname)) {      # for each (unique)
+ head, determine the cat_id that the head's item belongs to:
>     distinct.heads$headname[i] <- sub(x=distinct.heads$headname[i],
+     pattern="{",replacement="",fixed=T)
```

```
>     distinct.heads$headname[i] <- sub(x=distinct.heads$headname[i],
+     pattern="}",replacement="",fixed=T)
>     distinct.heads$headname[i] <- gsub(pattern="^ ",replacement="",
+     x=distinct.heads$headname[i],fixed=F)
>     distinct.heads$head_id[i] <- dbGetQuery(con.postgresql,paste("
+     SELECT cat_id FROM article_categories WHERE cat_desc LIKE
+     '%",distinct.heads$headname[i],"%'",sep=""))$cat_id[1]
> }
# insert each head and body id to its proper place in the rules table:
> for(i in 1:length(distinct.bodies$bodytext)) {
>     dbGetQuery(con.postgresql,paste("UPDATE ",rules.tablename," SET
+     bodyid='",distinct.bodies$body_id[i],"' WHERE bodytext =
+     '",distinct.bodies$bodytext[i],"'",sep=""))
> }
> for(i in 1:length(distinct.heads$headname)) {
>     dbGetQuery(con.postgresql,paste("UPDATE ",rules.tablename," SET
+     head='",distinct.heads$head_id[i],"' WHERE headname LIKE
+     '%",distinct.heads$headname[i],"%'",sep=""))
> }
```

Finally, the rule lengths need to be calculated and placed in the “length” column:

```
# calculating length of each rule:
> for(i in 1:length(rules.bodies)) {
>     rule.length <- length(do.call('rbind',strsplit(rules.bodies[i],
+     split=",",fixed=T))) + 1      # +1 because the rule length is
+     length(body) + length(head), and we know that head length is 1.
>     dbGetQuery(con.postgresql,paste("UPDATE ",rules.tablename,"
+     SET length='",rule.length,"' WHERE bodytext =
+     '",rules.bodies[i],"'",sep=""))
> }
```

The recommendations are now ready to be created. For the sakes of creating the recommendations, one table and four views are going to be created, which will contain the information needed in order to create the recommendations: the “rulebodies” table, which contains the body (or bodies) that each item belongs to, the “bodysizes” view, which shows the size of each body, the “bodies” view, which shows the body (or bodies) that the items belong to and the size of each body, the “bodies_trans” view, which shows the transaction and body (or bodies) that each item belongs to and the size of each body, and the bodies_trans_size view, which shows the transaction and body (or bodies) that each item belongs to, the size of each body, and how many items of each body that an item belongs to are contained to each transaction that the item in question belongs to.

The “rulebodies” table is created as follows:

```
> if(dbExistsTable(con.postgresql,"rulebodies"))
>     dbRemoveTable(con.postgresql,"rulebodies")
> dbGetQuery(con.postgresql,"CREATE TABLE rulebodies(bodyID
+ INTEGER,ITEMNAME VARCHAR(2500),ITEM INTEGER)")
> rulebodies.data <- dbGetQuery(con.postgresql,paste("SELECT DISTINCT
+ bodyid,bodytext FROM ",rules.tablename,sep=""))
> for(i in 1:length(rulebodies.data$bodyid)) {
>     current.rule.items <- do.call('rbind',
```

```
+     strsplit(rulebodies.data$bodytext[i],split=",",fixed=T))
>     for(j in 1:length(current.rule.items)) {
>           #trim '{}' and ending space from rule in order to search the
+           article_categories table for it:
>           current.rule.items[j] <- sub(x=current.rule.items[j],
+           pattern="{",replacement="",fixed=T)
>           current.rule.items[j] <- sub(x=current.rule.items[j],
+           pattern="}",replacement="",fixed=T)
>           current.rule.items[j] <- gsub(pattern=" $",
+           replacement="",x=current.rule.items[j],fixed=F)
>           #end trimming
>           item.details <- dbGetQuery(con.postgresql,paste("SELECT
+           cat_id,cat_desc FROM article_categories WHERE cat_desc LIKE
+           '%",current.rule.items[j],"%'",sep=""))
>           itemname <- item.details$cat_desc
>           item_id <- item.details$cat_id
>           dbGetQuery(con.postgresql,paste("INSERT INTO
+           rulebodies(bodyID,ITEMNAME,ITEM) VALUES('"
+           ,rulebodies.data$bodyid[i],"','",itemname,"',
+           '",item_id,"')",sep=""))
>     }
> }
```

The “bodysizes” view is created as follows:

```
>     dbGetQuery(con.postgresql,"CREATE VIEW bodysizes(bodyid, bodysize)
+     AS (select bodyid, count(*) from rulebodies group by bodyid)")
```

The “bodies” view is created as follows:

```
> dbGetQuery(con.postgresql,"CREATE VIEW bodies(bodyid, bodyitem,
+ bodysize) AS (select rb.bodyid, item, bodysize from rulebodies rb,
+ bodysizes bs where rb.bodyid=bs.bodyid)")
```

The “bodies_trans” view is created as follows:

```
> dbGetQuery(con.postgresql,"CREATE VIEW bodies_trans(trans_id, bodyid,
+ bodyitem, bodysize) as (select transid, bodyid, bodyitem, bodysize
+ from bodies, new_transactions where bodyitem = itemid)")
```

The information gained by the “bodies_trans” view can result in a first set of recommendations: the bodies having a bodysize of 1 consist only of the item being purchased by the transaction to which the recommendations are intended, therefore the heads of the bodies which bodysize is 1 comprise recommendations for that transaction.

After having recommended the heads of the rules whose bodies consist only of the item being purchased by the transaction to which the recommendations are intended, the rules containing more items need to be examined. This is achieved by creating the “bodies_trans_size” view. This way, information regarding how many items of each body are included in each transaction can be obtained. Therefore, the more items of a body a transaction contains, the more important

that rule is for that transaction, thus its head comprises a recommendation for that transaction.

The “bodies_trans_size” view is created as follows:

```
> dbGetQuery(con.postgresql,"CREATE VIEW bodies_trans_size(transid,
+ bodyid, bodysize, bodytranssize) as (select trans_id, bodyid,
+ max(bodysize), count(*) from bodies_trans group by bodyid, trans_id)")
```

Via the “bodies_trans_size” view, information about the rules whose bodies contain more than one item can be gained. The higher the value of the “bodytranssize” column, the more items of the transaction are included in the body of the rule, thus the more accurate recommendation the head of that rule is.

The final step is to gather all of the recommendations in one table. This way they can afterwards be further processed in whichever way the supermarket desires, for example they can be ordered by their lift metric value or their support metric value. The table is named “product_recommendations” and is constructed as follows:

```
> if(dbExistsTable(con.postgresql,"product_recommendations"))
>     dbRemoveTable(con.postgresql,"product_recommendations")
> dbGetQuery(con.postgresql,"CREATE TABLE product_recommendations
+ (transid INTEGER,recomm_item VARCHAR(30),support FLOAT,confidence
+ FLOAT,lift FLOAT)")
> dbGetQuery(con.postgresql,"insert into product_recommendations select
+ transid, headname, ar.support, ar.confidence, ar.lift from
+ bodies_trans_size bts,rules ar where ar.bodyid = bts.bodyid and
+ bts.bodysize = bts.bodytranssize")
```

Using the “product_recommendations” table the supermarket can henceforth accurately recommend items to customers, and therefore ensure a sufficiently high possibility of increment in its profit.

5.1.9 Discussion

Concluding, the relation among the items of the categories CONFECTIONERY, FRESH MEAT,SAUSAGES,FISH and MILK,CHEESE,EGGS seems to be highly important. Also, the items belonging to the categories FRESH MEAT,SAUSAGES,FISH, BACKED GOODS, SOUPS,SPICES,SPREAD and PROCESSED FOOD(EXC. SOUPS) seem to be highly related to the items of the category MILK,CHEESE,EGGS due to the fact that four of the most important rules of the mining contain MILK,CHEESE,EGGS in their RHS.

Comparing the results with the results produced in IBM’s tutorial [3], it is easily perceived that R can accurately produce the same results as IBM DB2 Intelligent Miner, since the results are exactly the same in both studies.

The association rules mined can be put to very many effective uses. A good and simple example of such a usage is the recommendation system demonstrated previously. Using a simple and logical statistical means, a supermarket can greatly increase its income by effectively and cleverly using information gained from an association rules mining process.

5.2 Classifying Titanic passengers using a decision tree

5.2.1 Introduction

The second case study demonstrates how classification of a dataset using the C4.5 algorithm to build a decision tree is performed in R. The dataset classified contains information about the passengers of Titanic, the British passenger liner that sank in the North Atlantic Ocean in 1912, and was retrieved from the datasets provided by "Data for Evaluating Learning in Valid Experiments" (DELVE) project of the Computer Science department of the University of Toronto [13]. The passengers are classified as whether they survived the tragedy or not, and the classification is made according to the passengers' sex, age and economical status (represented in this dataset by their ticket class)

5.2.2 Loading the required packages into R

To begin the classification procedure, the first thing required to do is to load all of the packages that are going to be used:

```
> library("RWeka")
> library("party")
> library("partykit")
> library("FSelector")

Loading required package: zoo

Attaching package: 'zoo'

The following object is masked from 'package:base':

    as.Date, as.Date.numeric

Loading required package: sandwich
Loading required package: strucchange
Loading required package: modeltools
Loading required package: stats4

Attaching package: 'modeltools'

The following object is masked from 'package:igraph':

    clusters
```

```
The following object is masked from ‘package:arules’:

    info


Attaching package: ‘partykit’

The following object is masked from ‘package:party’:

    ctree, ctree_control, edge_simple, node_barplot, node_boxplot,
node_inner, node_surv, node_terminal
```

If an output like the above is displayed and no errors occur, the packages have been successfully loaded into R.

5.2.3 Reading the passengers information into R

The second step to take is to read the dataset information into R:

```
> titanic.dataset <-
+ read.table("Dataset.data",header=F,sep=",",quote="",
+ stringsAsFactors=T,blank.lines.skip=T,allowEscapes=F,
+ colClasses=c("factor","factor","factor","factor"),
+ col.names=c("class","age","sex","survived"))
```

All of the function parameters are explained in the previous case study, except for the ‘stringsAsFactors’ parameter, which instructs the function to store any character data read from the dataset in a factor. And since all of the data in this case is categorical, is needs to be stored in factors – as indicated in the ‘colClasses’ parameter.

5.2.4 Building the decision tree

Once having successfully read the data from the dataset, the decision tree is ready to be built. To build it, the ‘J48()’ function is used:

```
> tree <- J48(survived~.,data=titanic.dataset)
```

Parameters explanation:

- ‘survived~.’ – instructs the algorithm to classify the data into classes of the “survived” factor, and to categorize the data according to all of the other attributes (expressed with “.”). The tilde (“~”) symbol is used merely as a delimiter between the factor to categorize as and the factors to categorize by.

- ‘data=titanic.dataset’ – indicates the data source.

5.2.5 Visualizing and evaluating the results

Once successfully building the decision tree, its results can be viewed by the name of the object that the tree was stored into:

```
> tree

J48 pruned tree
------------------

sex = female
|   class = 1st: yes (145.0/4.0)
|   class = 2nd: yes (106.0/13.0)
|   class = 3rd: no (196.0/90.0)
|   class = crew: yes (23.0/3.0)
sex = male
|   class = 1st
|   |   age = adult: no (175.0/57.0)
|   |   age = child: yes (5.0)
|   class = 2nd
|   |   age = adult: no (168.0/14.0)
|   |   age = child: yes (11.0)
|   class = 3rd: no (510.0/88.0)
|   class = crew: no (862.0/192.0)

Number of Leaves  :     10

Size of the tree :      15
```

Observing the resulting decision tree, it can be concluded that the attribute picked by the algorithm to be placed in the root node was the “sex” attribute, because it has the highest information gain among the 3 attributes:

```
> information.gain(survived~.,data=titanic.dataset)

      attr_importance
class     0.059287937
age       0.006410718
sex       0.142391195
```

It is perceivable that the female passengers are not classified by age, as are the male passengers. The error rates of this decision tree can be viewed in its confusion matrix:

```
> summary(tree)

=== Summary ===

Correctly Classified Instances        1740               79.055  %
Incorrectly Classified Instances       461               20.945  %
Kappa statistic                          0.4334
```

```
Mean absolute error                      0.3089
Root mean squared error                  0.393
Relative absolute error                 70.6078 %
Root relative squared error             84.0339 %
Coverage of cases (0.95 level)          99.8183 %
Mean rel. region size (0.95 level)      96.3426 %
Total Number of Instances             2201

=== Confusion Matrix ===

    a    b   <-- classified as
 1470   20 |    a = no
  441  270 |    b = yes
```

Observing the above output, among a total of 2201 instances the tree correctly classified 1740 of them, which correspond to a percentage of 79.055% of the total instances, whereas it incorrectly classified 461 instances, which correspond to a percentage of 20.945% of the total instances. Observing the confusion matrix, it seems that the tree has real trouble in correctly classifying the survivals, since it has classified more than half of them as non-survivals.

An alternative representation of the same tree, where the error rates for each leaf are visible:

```
> as.party(tree)

Model formula:
survived ~ class + age + sex

Fitted party:
[1] root
|   [2] sex in female
|   |   [3] class in 1st: yes (n = 145, err = 2.8%)
|   |   [4] class in 2nd: yes (n = 106, err = 12.3%)
|   |   [5] class in 3rd: no (n = 196, err = 45.9%)
|   |   [6] class in crew: yes (n = 23, err = 13.0%)
|   [7] sex in male
|   |   [8] class in 1st
|   |   |   [9] age in adult: no (n = 175, err = 32.6%)
|   |   |   [10] age in child: yes (n = 5, err = 0.0%)
|   |   [11] class in 2nd
|   |   |   [12] age in adult: no (n = 168, err = 8.3%)
|   |   |   [13] age in child: yes (n = 11, err = 0.0%)
|   |   [14] class in 3rd: no (n = 510, err = 17.3%)
|   |   [15] class in crew: no (n = 862, err = 22.3%)


Number of inner nodes:     5

Number of terminal nodes: 10
```

To create a visual representation of the resulting decision tree, the “plot()” function needs to be used. And because the tree contains multi-way splits, it needs to be coerced into a “party” class object in order to be plotted successfully:

```
> plot(as.party(tree))
```

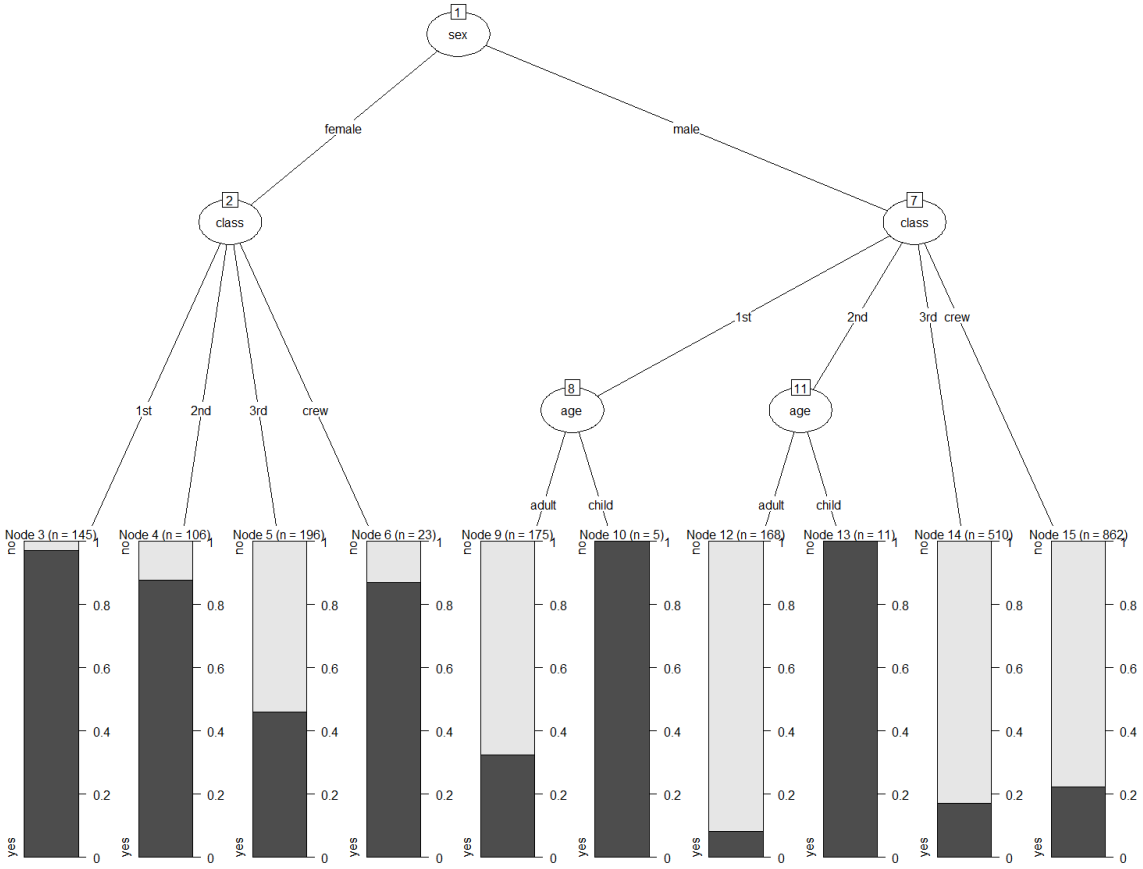

Figure 5.2.5-1: Detailed visual representation of the decision tree

As is easily perceived, the 3rd class passengers have the lowest survivability, while the female passengers in general have the highest. Another interesting fact is that all of the 1st and 2nd class children survived. Passengers belonging to the 3rd class seem to be the most doomed – out of 706 people, 528 died.

5.2.6 Discussion

Observing the outputs, it can be concluded that most of the crew didn’t make it (only 212 survivals out of 885 crew members) and about half of the children didn’t survive as well (57 survived out of a total of 109 children). Regarding the rest of the passengers, a total of 296 women survived out of 402, but only 146 male passengers made it out of a total of 805. Also, the higher the class, the higher the

survival rate tends to be (about more than half of the passengers of the 1st and 2nd classes survived, while only 178 3rd class passengers survived out of 706). Regarding the decision tree performance in classifying the passengers, the error rate was ~21%.

6. Epilogue

This study was conducted in order to explore the capabilities of the R language with regards to data mining and to connecting to, and exchanging information with, popular DBMSs such as MySQL and PostgreSQL. After thoroughly studying the language and its capabilities and conducting two case studies in which the data mining capabilities of the language using the association rules mining and classification methods were examined and evaluated, the can conclusion that can be drawn is that the R programming language's scalability with regards to data mining is great, as is its extensionality ability and also the speed it achieves when performing complex data mining calculations and model constructions. With little effort, a thorough statistical analysis and many data mining techniques can be accurately performed on information stored in databases, the results of which can be sufficiently visualized in order to allow information to be obtained by them and permanently stored outside the R environment, thus becoming available for further processing and manipulation even after the R session is ended.

Finally, as a suggestion on the continuation of the case studies conducted for the sakes of this thesis, a clustering analysis using the "Nearest Neighbor" algorithm could be performed, in order to observe whether an unsupervised categorization of the data will be as accurate as the supervised categorization conducted in the second case study of this thesis using the "Classification" data mining method and utilizing the "C4.5" decision tree constructing algorithm.

7. References

[1] DBTechNet. Retrieved November, 2013, from http://dbtechnet.org/

[2] Genolini, C. (2008), “A (not so) short introduction to S4: Object Programming in R”. Retrieved November, 2013, from http://cran.r-project.org/doc/contrib/Genolini-S4tutorialV0-5en.pdf

[3] IBM, “Mining your Business in Retail with IBM DB2 Intelligent Miner”. Retrieved November, 2013, from http://www.ibm.com/developerworks/edu/dm-dw-dm-retail_tutorial-i.html

[4] Kumar, R et al (2012), "Classification Algorithms for Data Mining: A Survey", Department of Computer Science and Engineering, Jind Institute of Engg. & Technolog, Jind, Haryana, India, August 2012. Retrieved November, 2013, from http://ijiet.com/wp-content/uploads/2012/09/2-1.pdf

[5] Palace, B. (1996), “Data Mining, a technology note prepared for Management 274A, Anderson Graduate School of Management at UCLA, Spring 1996”. Retrieved November, 2013, from http://www.anderson.ucla.edu/faculty/jason.frand/teacher/technologies/palace/datamining.htm

[6] The Comprehensive R Archive Network, “arules: Mining Association Rules and Frequent Itemsets”. Retrieved November, 2013, from http://cran.r-project.org/web/packages/arules/index.html

[7] The Comprehensive R Archive Network, “arulesViz: Visualizing Association Rules and Frequent Itemsets”. Retrieved November, 2013, from http://cran.r-project.org/web/packages/arulesViz/index.html

[8] The Comprehensive R Archive Network, “Kickstarting R”. Retrieved November, 2013, from http://cran.r-project.org/doc/contrib/Lemon-kickstart/kr_rfunc.html

[9] The Comprehensive R Archive Network, “Mining Association Rules and Frequent Itemsets”. Retrieved November, 2013, from http://cran.r-project.org/web/packages/arules/arules.pdf

[10] The R Project for Statistical Computing, “What is R?”. Retrieved November, 2013, from http://www.r-project.org/about.html

[11] Togaware, “Rattle: A Graphical User Interface for Data Mining using R”. Retrieved November, 2013, from http://rattle.togaware.com/

[12] Torgo, L. (2011), “Data Mining With R: Learning with case studies”. Published November 19, 2010, by Chapman & Hall/CRC. Retrieved April, 2013, from http://irandataminer.ir/files/book/DataMining-with-R-English.pdf

[13] University of Toronto, Department of Computer Science, “Data for Evaluating Learning in Valid Experiments”. Retrieved November, 2013, from http://www.cs.utoronto.ca/~delve/

[14] Wikibooks, open books for an open world, “R Programming/Introduction”. Retrieved November, 2013, from https://en.wikibooks.org/wiki/R_Programming/Introduction

[15] Wikipedia: The free encyclopedia. (2013, November). FL: Wikimedia Foundation, Inc., “Apriori algorithm”. Retrieved November, 2013, from https://en.wikipedia.org/wiki/Apriori_algorithm

[16] Wikipedia: The free encyclopedia. (2013, November). FL: Wikimedia Foundation, Inc., “Association rule learning”. Retrieved November, 2013, from https://en.wikipedia.org/wiki/Association_rule_learning

[17] Wikipedia: The free encyclopedia. (2013, November). FL: Wikimedia Foundation, Inc., "Data mining”. Retrieved November, 2013, from https://en.wikipedia.org/wiki/Data_mining

[18] Wikipedia: The free encyclopedia. (2013, November). FL: Wikimedia Foundation, Inc., “Database”. Retrieved November, 2013, from https://en.wikipedia.org/wiki/Database

[19] Wikipedia: The free encyclopedia. (2013, November). FL: Wikimedia Foundation, Inc., “Lift (data mining)”. Retrieved November, 2013, from https://en.wikipedia.org/wiki/Lift_(data_mining)

[20] Wikipedia: The free encyclopedia. (2013, November). FL: Wikimedia Foundation, Inc., “MySQL”. Retrieved November, 2013, from https://en.wikipedia.org/wiki/MySQL

[21] Wikipedia: The free encyclopedia. (2013, November). FL: Wikimedia Foundation, Inc., “PostgreSQL”. Retrieved November, 2013, from https://en.wikipedia.org/wiki/PostgreSQL

[22] Wikipedia: The free encyclopedia. (2013, November). FL: Wikimedia Foundation, Inc., “Pruning (decision trees)”. Retrieved November, 2013, from https://en.wikipedia.org/wiki/Pruning_(decision_trees)

[23] Wikipedia: The free encyclopedia. (2013, November). FL: Wikimedia Foundation, Inc., “R (programming language)”. Retrieved November, 2013, from https://en.wikipedia.org/wiki/R_(programming_language)

[24] Wikipedia: The free encyclopedia. (2013, November). FL: Wikimedia Foundation, Inc., “Rattle GUI”. Retrieved November, 2013, from https://en.wikipedia.org/wiki/Rattle_GUI

[25] Wikipedia: The free encyclopedia. (2013, November). FL: Wikimedia Foundation, Inc., “Supervised learning”. Retrieved November, 2013, from https://en.wikipedia.org/wiki/Supervised_learning

[26] Williams, G. (2009), “Rattle: A Data Mining GUI for R”. Retrieved November, 2013, from http://journal.r-project.org/archive/2009-2/RJournal_2009-2_Williams.pdf

[27] Wing, K. et al (2013), "Data Mining: An Introduction", University of North Carolina at Chapel Hill. Retrieved November, 2013, from http://www.unc.edu/~xluan/258/datamining.html

[28] Yau, C. (2013), “R Tutorial An R Introduction to Statistics”. Retrieved November, 2013, from http://www.r-tutor.com/r-introduction/data-frame

[29] Zhao, Q et al (2003), “Association Rule Mining: A Survey”, Nanyang Technological University, Singapore. Retrieved November, 2013, from http://sci2s.ugr.es/keel/pdf/specific/report/zhao03ars.pdf

[30] Zhao, Y. (2012), “R and Data Mining: Examples and Case Studies”. Retrieved November, 2013, from http://www.rdatamining.com/docs/RDataMining.pdf

9. Appendix

9.1 Rattle

Rattle GUI is a free and open source software package providing a graphical user interface (GUI) for performing data mining using R. It is currently used around the world in a variety of situations – currently 15 different government departments in Australia and around the world use Rattle in their data mining activities, and also as a statistical package. Rattle provides considerable data mining functionality by exposing the power of the R Statistical Software through a graphical user interface [24].

Its purpose is to ease the use of data mining in R, simplifying many of the actions needed to be taken in order for R to be able to perform data mining on a dataset. The aim is to provide an intuitive interface that takes you through the basic steps of data mining, as well as illustrating the R code that is used to achieve this. Whilst the tool itself may be sufficient for all of a user's needs, it also provides a stepping stone to more sophisticated processing and modeling in R itself, for sophisticated and unconstrained data mining [11].

Rattle is also used as a teaching facility to learn the R language as well. There is a code tab which replicates the R code for any activity undertaken in the GUI, which can be copied and pasted.

Rattle can also be used for statistical analysis or model generation. It allows the dataset to be partitioned into training, validation and testing. The dataset can be viewed and edited within the Rattle GUI [24].

Its interface is tab-oriented; Rattle accepts input from files, and its capabilities include the following: Calculation of statistical metrics, capability of carrying out various statistical tests, ability of performing data mining methods such as clustering or classification, ability of visualizing the results and constructing the relevant models (such as decision trees, for example), ability of building graphical charts such as histograms or bar charts to effectively represent these results.

Rattle uses the Gnome graphical user interface as provided through the RGtk2 package (Lawrence and Lang, 2006). It runs under various operating systems, including GNU/Linux, Macintosh OS/X, and MS/Windows [26].

9.2 Core (most useful / mostly used) functions

The following is a list of some of the most common functions that are mostly used in R:

- seq() – generates a sequence (for example, a mathematical sequence)

- rep() – repeats a supplied pattern
- length() – returns the length of an object
- mean() – calculates and returns the arithmetic mean of the components of an R object
- var() – calculates and returns the variance of the components of an R object
- dim() – prints the dimensions of an object
- typeof() – prints the type of an object
- class() – prints the class of an object
- str() – prints the internal structure of an R object
- library() – loads a package
- factor() – transforms a vector into a factor
- example() – very useful function, prints an example of the appropriate usage of the specified command or function