Master Thesis in

Computational Engineering and Intelligent Systems

Universidad del País Vasco Euskal Herriko Unibertsitatea

Konputazio Zientziak eta Adimen Artifiziala Saila –
Departamento de Ciencias de la Computación e Inteligencia Artificial

Master C.E.I.S.

Preprocess and Data Analysis Techniques for Affymetrix DNA Microarrays Using Bioconductor: A Case Study in Alzheimer disease

Alberto Poncelas

Supervisor

Iñaki Inza
Department of Computer Science and Artificial Intelligence
Computer Science Faculty

informatika fakultatea facultad de informática

KZAA /CCIA

June 2013

Abstract

DNA microarray, or DNA chip, is a technology that allows us to obtain the expression level of many genes in a single experiment. The fact that numerical expression values can be easily obtained gives us the possibility to use multiple statistical techniques of data analysis.

In this project microarray data is obtained from Gene Expression Omnibus, the repository of National Center for Biotechnology Information (NCBI). Then, the noise is removed and data is normalized, also we use hypothesis tests to find the most relevant genes that may be involved in a disease and use machine learning methods like KNN, Random Forest or Kmeans.

For performing the analysis we use Bioconductor, packages in R for the analysis of biological data, and we conduct a case study in Alzheimer disease. The complete code can be found in `https://github.com/alberto-poncelas/bioc-alzheimer`

Contents

1 Introduction **1**

2 Molecular biology and microarrays **3**
2.1 Introduction to molecular biology 3
2.1.1 Molecules involved in life 3
2.1.2 DNA and RNA 4
2.2 Microarray technology 5

3 Overview of the steps in the analysis of microarray data **7**
3.1 The pipeline 7
3.2 Gene Expression Omnibus 8
3.3 Bioconductor 10
3.3.1 Starting with Bioconductor 10
3.3.2 Bioconductor objects for gene expression 10
3.3.3 Finding gene information 11

4 Preprocess techniques for raw signals **13**
4.1 Background correction 13
4.1.1 RMA background correction 14
4.1.2 MAS 5.0 background subtraction 14
4.2 Normalization methods 15
4.2.1 Scaling 15
4.2.2 Quantile normalization 15
4.2.3 Variance Stabilization and Normalization (VSN) 16
4.3 Summarization 17
4.3.1 Tukey Bi-Wight 17
4.3.2 Medianpolish 18
4.4 Probe correction 19

5 Quality assessment of microarray data **21**
5.1 Quality assessment plots 21
5.1.1 Probe intensities: density histograms and boxplot 21
5.1.2 QCstats 22
5.1.3 RLE and NUSE plots 24

5.1.4 MA plots . . . 26
5.1.5 RNA degradation . . . 27

6 Methods for filtering genes from an ExpressionSet 29
6.1 Filter genes by variance . . . 29
6.1.1 Non-Specific filter . . . 29
6.2 Filter genes by means of parametric statistical tests . . . 30
6.2.1 Mean difference t-test . . . 30
6.2.2 Multiple testing . . . 31
6.2.3 Moderated t-statistics . . . 32

7 Supervised and unsupervised learning techniques for the analysis of microarray data 35
7.1 Preparing data . . . 35
7.1.1 Standardization . . . 35
7.1.2 Distance matrix . . . 36
7.1.3 Principal component analysis (PCA) . . . 36
7.2 Supervised learning (Classification) . . . 36
7.2.1 KNN . . . 37
7.2.2 Random forest . . . 37
7.3 Unsupervised learning (Clustering) . . . 38
7.3.1 K-means . . . 38
7.3.2 Hierarchical clustering . . . 39
7.4 Evaluation . . . 39
7.4.1 Cross validation (for classification) . . . 39
7.4.2 Silhouette (for clustering) . . . 41

8 A Case study on microarray data analysis for Alzheimer disease 43
8.1 The data set . . . 43
8.2 Import data . . . 44
8.3 Preprocess data . . . 45
8.4 Filter genes . . . 49
8.5 Classification . . . 51
8.6 Clustering . . . 53
8.7 Obtaining information about genes . . . 57

A R Code 63
A.1 Helper functions . . . 63
A.2 Main script . . . 66

Chapter 1

Introduction

The main motivation of this project is to create a manual with a recopilation of the most used techniques for the analysis of gene expression level in microarray technology. Another aim is to explain what Bioconductor is, and how to use it to perform in R the whole data analysis process applying the techniques explained in the document.

For the good comprehension of the project this document is structured as follows.

In the first chapter we give an introduction about biology. We focus specially in DNA and how the proteins are created. Also we introduce the microarray technology.

In the next chapter we give an overview of the necessary steps for a whole data analysis process. We also explain what Gene Expression Omnibus is and how to obtain data from there. Also we introduce Bioconductor and give a sample script with the first steps to start using it.

Once Bioconductor is installed and we have imported the raw data, we need to preprocess it, that is, remove noise, normalize, etc. We explain some of the most used techniques to do that.

Also, in order to control the quality of the data before, meanwhile and after preprocessing it we need a way of visualizing the data, displaying for that different plots. We mention some of the most used ones.

When analysing a particular disease we are only interested in genes which are differentially expressed. If a gene shows similar expression levels for a disease samples and control samples it means that it does not have an influence on the disease. We want to remove these genes and keep only the ones which are differentially expressed. For doing that we can use statistical techniques, we

mention some of them.

In addition, we also introduce the concept of supervised and unsupervised learning, give some examples of them and explain some ways to validate the quality of the models produced.

In the last chapter, we do a case study in Alzheimer disease. We take both samples of control and severe stage of Alzheimer and perform a whole data analysis using mentioned techniques in this document

Chapter 2

Molecular biology and microarrays

For a better understanding it is useful to have a basic knowledge on molecular biology and microarray technology. The purpose of this chapter is to give an overview of them.

2.1 Introduction to molecular biology

In this section we are going to give a brief explanation about molecular biology, with an emphasis on molecules of DNA, RNA and the process of protein creation. The purpose of this is to give an introduction to people without a background in biology.

2.1.1 Molecules involved in life

There are four kind of molecules involved in life [1]:

- *Small molecules*: Those molecules have different roles such as being a source of energy, signal transmission or being the building blocks of other molecules. Some important building blocks are nucleotides and amino acids. DNA and RNA are made of nucleotides and proteins are made of amino acids.
- *Proteins*: These are the main building blocks and functional molecules of the cell. They have multiple functions like catalysing biochemical reactions, cell signalling or being a building block of different organisms.
- *DNA*: Deoxyribonucleic acid, or DNA, is the molecule that stores all the information to create proteins.
- *RNA*: Ribonucleic acid, or RNA, is a molecule that takes part in the process of protein synthesis, using the information in DNA.

2.1.2 DNA and RNA

As mentioned before, DNA is the molecule that stores all the information to create proteins and it can be found in the nucleus of every cell of every organism.

The structure of DNA is usually represented as a two-stranded chemical structure, which is called double helix. Each of these strands is a chain of molecules called nucleotides. There are four types of nucleotides, depending on the chemical base they have: adenosine, guanine, cytosine and thymine. They are usually denoted by their initial letters A, G, C and T. To represent a strand of nucleotides we use those letters, and at the ends of the sequence, 5' and 3', by convention it is usually written with 5' on the left and 3' on the right:

```
5'   A-G-T-C-C-A-A-G-C-T-T   3'
```

In double stranded DNA, the two strands are paired forming bounds between them:

```
5'  C-G-A-T-T-G-C-A-A-C-G-A-T-G-C   3'
    | | | | | | | | | | | | | | |
3'  G-C-T-A-A-C-G-T-T-G-C-T-A-C-G   5'
```

Only two kind of bounds are possible: A-T and C-G. This means that two strands are complementary, one strand fully determines the other one.

Because of this property DNA replication is possible (this happens for example, during the cell division). During this process the DNA double helix unwinds and forks. Then, from each strand, their complimentary strand is synthesised. After the process, there are two identical double-helix DNA molecules.

RNA is constructed from nucleotides (as DNA is), but instead of having the base thymine (T), it has uracil (U). RNA are single stranded (they do not form a double helix) and can be complementary to a single strand of a DNA molecule and bind to it but, instead A-T bounds, A-U bounds are created.

```
C-G-A-T-T-G-C-A-A-C-G-A-T-G-C        (DNA)
| | | | | | | | | | | | | | |
G-C-U-A-A-C-G-U-U-G-C-U-A-C-G        (RNA)
```

There are three kind of RNA: messenger RNA (mRNA), ribosomal RNA (rRNA), and transfer RNA (tRNA). Messenger RNA is the one that plays a major role in microarray experiments [5] and its function is to carry the genetic information from DNA so proteins can be synthesized.

The process of making proteins is called protein synthesis and it has three phases:

1. Transcription: In this phase one strand of DNA is copied into a complementary pre mRNA.

2. Splicing: the pre mRNA created in the previous step has parts which code proteins (Exons) and parts which do not (Introns). Exons and Introns are interspersed. The splicing phase consists on removing Introns and joining Exons together. The result of this step is mRNA (from pre mRNA to mRNA).

3. Translation: In this phase the proteins are created. These proteins are made by joining together the amino acids encoded by mRNA. Only 20 different amino acids exist, each one encoded by a triplet (three adjacent nucleotides of mRNA) called codon. Since there are four different nucleotides (A, G, C or T) a triplet is required to encode a amino acid (because $4^3 > 20$).

2.2 Microarray technology

The microarrays we are using in this project are one-channel (the arrays provide the expression level, two-channel arrays compares two DNA samples) microarrays from Affymetrix. Affymetrix, `http://www.affymetrix.com/`, is an American company, based in California, that manufactures DNA microarrays.

A microarray, or DNA chip, is a glass or polymer slide where DNA molecules are attached at fixed locations (spots). Each gene and expressed sequence tag is represented by oligonucleotides on the array which matches the sequence of the gen of interest [6]. Each of this oligonucleotides is called probe. The purpose is to measure the abundance of labeled mRNA obtained from biological samples. For each transcript there are a group of probes called probe set.

Figure 2.1: Illustration of a probe set. It consist on multiple pairs of PM and MM

A probe set consist on two types of probes: PM which measures the hybridization level, and MM (Mismatch) which measures cross-hybridization level, hybridization in a probe that is supposed to detect another mRNA. Usually PM and MM probes are together (so the background effect is similar) so we refer to them as probe pairs. In each data set there are between 11 and 20 probe pairs [6, 7].

For obtaining gene expression levels we need to do these steps [2]:

1. Extract DNA. It can be obtained from any biological sample such as blood or saliva.

2. DNA is processed to obtain millions of short pieces. Also to each piece is attached a molecule called biotin.

3. The prepared DNA sample is placed over the array for 14 to 16 hours. During that time some DNA slices will attach to microarray.

4. After that a fluorescent molecule placed on the microarray. This fluorescent molecule will attach to biotin. This means that fluorescent will be attached to DNA slices that are attached to microarray.

5. The microarray is read by a laser and the expression levels are obtained depending on the quantity of the fluorescent molecule.

Chapter 3

Overview of the steps in the analysis of microarray data

In this chapter we want to give an overview of a complete microarray data analysis and a small guide to how to start. We explain the steps done to go from raw data to machine learning models. The tool we are using to do this is Bioconductor, an open source project in R for genomic analysis. In this chapter we also provide an introduction to install and start using it.

3.1 The pipeline

In the Figure 3.1 we show which is the pipeline we are going to follow on this document:

- First we import the raw data, which is stored in files with .CEL extension. Each file contains the data of one sample. Imported files will be stored in memory in an *AffyBatch* object.
- The data of *AffyBatch* object must be preprocessed and normalized. We also have some techniques so we can asses the quality of data, and remove outliers if needed, or help us deciding wether to use different preprocessing techniques or not.
- Once the data has been conveniently preprocessed it is converted into an *ExpressionSet* object. We are going to use this object to create models
- A step we can do with the *ExpressionSet* is a feature selection. We can try to remove the genes which are not informative for a particular disease.
- Finally we can use different machine learning models. In this document we explain some classification and clustering techniques.

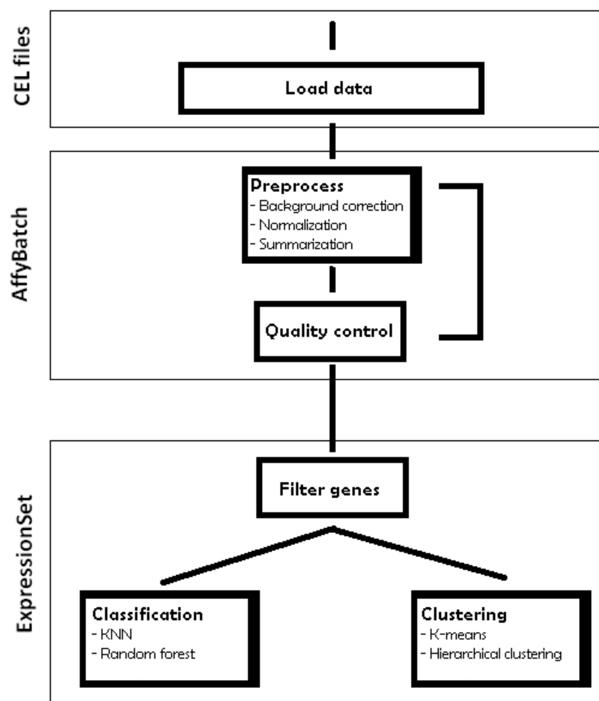


Figure 3.1: The pipeline with the steps of the complete microarray data analysis that we are following on this document.

3.2 Gene Expression Omnibus

The microarray data used for this project has been obtained from Gene Expression Omnibus (GEO). Gene Expression Omnibus is a public functional genomics data repository [3] from NCBI (National Center for Biotechnology Information).

For searching sets of microarray data it is possible to access to GEO DataSet browser, `http://www.ncbi.nlm.nih.gov/sites/GDSbrowser/`,

Figure 3.2: Main screen of GEO DataSet browser [3]

In the Figure 3.2 we present GEO DataSet browser main screen. The main three parts are:

1. A header with the logos and a search box field which can be used for searching for datasets.

2. A table listing the datasets. This is where results of the search are displayed. Clicking on the "Series" code it is possible to access to the experiment page, where an experiment summary and other information is displayed. This is specially interesting for us, because microarray raw data (.CEL files) may be provided in that site.

3. A table with information of selected data set. Here is specially interesting for us is the button of the top called "Sample Subsets". There we can see the information about each sample of the dataset. Also, on the right, we can download the dataset already processed in several formats.

3.3 Bioconductor

In this section we are going to explain how to start using Bioconductor. Bioconductor is an open source, open development software project, based primarily on the R programming language, which provide tools for the analysis of genomic data [4].

3.3.1 Starting with Bioconductor

To start using Bioconductor, we need first to install (or update) it. We can do it executing the following code:

```
source("http://bioconductor.org/biocLite.R")
old.packages(repos=biocinstallRepos())
library("Biobase")

#To install a library, execute the following command:
#biocLite("LIBRARY_NAME")
```

If we need to use a Bioconductor library that we have not installed yet, we need to execute `biocLite` command to install it (once *Biobase* library is loaded). For example, to install *affy* library, we need to execute `biocLite("affy")`.

Once Bioconductor is installed, we can load microarray files (like CEL files). To read every file in the current folder, we execute the following:

```
library("affy")
library("AnnotationDbi")
AffyBatchObject = ReadAffy()
```

3.3.2 Bioconductor objects for gene expression

In this section we explain the two main objects we are going to use: the *AffyBatch* object and the *ExpressionSet*. These objects contain the all data of microarray (expression levels, information about the samples...) in a single object. The main difference between them is that *AffyBatch* stores raw data and *ExpressionSet* the data after being preprocessed. Once we have applied preprocessing techniques and obtained the *ExpressionSet* we are ready to perform different data analysis techniques.

The *ExpressionSet* and *AffyBatch* have a similar structure:

- Assay data: It contains the matrix of expression values obtained from microarrays.
- Sample annotation: It contains the information about the samples (age,sex, treatment status...)

- Feature data: It contains the description of the variables of the sample annotation.
- Experiment data: It contains information about the laboratory, investigators etc. where the experiment was done.

To access to each field of the Expression set:

```
#Obtain the assayData:
exprs(ExpressionSet)

#Obtain the Sample annotation:
pData(ExpressionSet)

#Obtain the description of the features:
varMetadata(ExpressionSet)

#Obtain the Experiment data:
experimentData(ExpressionSet)
```

Also the following functions are useful:

```
# Probe names
featureNames(ExpressionSet)

# Sample names
sampleNames(ExpressionSet)

# Columns of Sample annotation
varLabels(ExpressionSet)
```

3.3.3 Finding gene information

The *ExpressionSet* object stores the expression levels of the probes. We may want to know which is the gene that represents a particular probe. Information about genes can be found in the NCBI gene database `http://www.ncbi.nlm.nih.gov/gene`.

For obtaining the corresponding gene of a probe we can create a table executing this code:

```
library("hgu95av2.db")
#genes database
genesDB<-merge(
        toTable(hgu95av2ENTREZID),
        toTable(hgu95av2GENENAME))
```

We can have an overview of the generated table by displaying first rows, executing `head(entrezGenesDB)`. Then, to obtain more information about a particular gene we can search their id or name in the NCBI gene database.

Chapter 4

Preprocess techniques for raw signals

In this chapter we explain some preprocessing techniques. This techniques are needed so we can compare different microarrays. They are divided in 3 groups, and usually (although there are exceptions) are used in the following order:

1. Background correction: Elimination of background noise.
2. Normalization: Scaling the different microarrays so they can be comparable.
3. Summarization: Obtaining a single value for each gen from the values of different probes.

We will focus on the techniques of following preprocessing procedures:

- MAS 5.0 (MAS+Scaling+Tukey). It is the preprocessing procedures of Affymetrix. It does and adjust of PM and MM [8]. In this preprocessing process summarization step is done before normalization.
- RMA (RMA+QuantileNorm+Medianpolish). Uses only PM values (and ignores MM) [5].

From now on we use, as notation, i to denote probes and j to denote microarrays.

4.1 Background correction

When analysing microarrays, the measure obtained from the probes are usually affected by the background, which is considered as a kind of noise. The methods of background correction try to estimate how much does the background signal affect to the signal so we can subtract it.

```
library("affyPLM")
#Methods: "MAS", "RMA.2" among others
preprocess(AffyBatchObject,
        background.method="MAS",
)
```

4.1.1 RMA background correction

The RMA method uses only PM probes and ignore MM. This method models the observed intensity as $O = S + N$. The intensity of the observation consist on a signal component S which is the true signal value and assumes it has a $Exp(\alpha)$ distribution, and a component of noise N with a normal distribution $N(\mu, \sigma^2)$.

In this process we try to estimate the true value S given an observation o. For that, [9] proposes the following estimator model:

$$E(s|O = o) = a + \sigma \frac{\phi(\frac{a}{\sigma}) - \phi(\frac{o-a}{\sigma})}{\Phi(\frac{a}{\sigma}) + \Phi(\frac{o-a}{\sigma}) - 1}$$

where:

- Φ the normal distribution function: $\Phi(z) = \int_{-\infty}^{z} \frac{1}{\sqrt{2\pi}} exp(-\frac{1}{2}w^2)dw$
- ϕ the density function: $\phi(z) = \frac{1}{\sqrt{2\pi}} exp(-\frac{1}{2}z^2)$
- $a = o - \mu - \sigma^2\alpha$

It is assumed that values of μ, α y σ^2 are the same for every PM probe intensity of every chip.

4.1.2 MAS 5.0 background subtraction

The algorithm of background correction MAS is the one proposed by Affymetrix. To estimate the background level it considers to the position of each probe in the chip [5, 8]. For subtracting the background level and calculate the new intensities we do the following steps:

1. The chip is split up in K different zones (16 by default).
2. For each zone K it is calculated the background (b_k) and the noise (n_k)
    (a) The average of 2% lowest intensities is chosen as background (b_k).
    (b) The standard deviation of the 2% lowest is chosen as noise (n_k)
3. For each element (x, y) it is calculated the background level $b(x, y)$. For that it is used a weighted average of different b_k. As weight, the distance between the element and the centroid of zone k is used.

4. In the same way we compute $n(x, y)$ for each element calculating a weighted average of the differents n_k.

5. Finally, the Background adjusted intensity is given:

$$I_{new}(x, y) = max\{I(x, y) - b(x, y), 0.5 * n(x, y)\}.$$

4.2 Normalization methods

Normalization is the process of removing unwanted non-biological variation that might exist between microarrays [9]. To make measurements from different microarrays to be comparable we need to eliminate differences caused by the different amount of RNA, scanner settings, etc. For this reason, we need a method to bring the arrays onto a similar scale.

```
library("affyPLM")
#Methods: "quantile", "scaling" among others
preprocess(AffyBatchObject,
        normalize.method="scaling"
)
```

4.2.1 Scaling

Scaling is a normalization method to transform the data so every microarray have the same statistical measure such as mean, median or, as it is the case of MAS 5.0, the 2% trimmed mean, the mean after removing the lowest and highest 2%.

The intensities of each probe is multiplied by a scaling factor f_j, different for each array j. The value of the factor is calculated [8]:

$$f_j = \frac{S_c}{TrimMean(S_j, 0.2, 0.98)}$$

The value of S_c is 500 by default. $TrimMean(S_j, 0.2, 0.98)$ is the mean of the intensities of array after truncating the highest and lowest 2% intensities (that is the reason for 0.2 and 0.98 values).

4.2.2 Quantile normalization

Quantile normalization is a normalization method that makes identical the distributions of intensities of multiple arrays.

1. The highest values of each array are chosen.

2. The mean of those values is calculated.

3. In the array, the values chosen are substituted by the obtained mean.
4. The steps 1 and 2 are repeated, but choosing the second highest values, then the same thing with third highest values... until we do the same with every value of the chip.

After the quantile normalization is applied, in every array there will be the same values. Obviously, thos values will not be on the same positions. Each array will have their highest values in the same positions the highest values were before normalization, and the same thing with the lowest values. However, after quantile normalization, every array will have the same distribution, the same mean, median...

4.2.3 Variance Stabilization and Normalization (VSN)

For normalizing Huber et al. [10] propose the following model:

$$y_{ij} = \alpha_{ij} + \beta_{ij} x_{ij}$$

Where x_{ii} represents the true expression value, β_{ij} is a scaling factor and α_{ij} is the chip offset.

In the model, a part of α_{ij} and β_{ij} is the same within a chip j, so we can express it as:

- $\beta_{ij} = \beta_j \gamma_{ij} e^{\eta ij}$, where β_j is a scale factor, the same in the chip j, and γ_{ij} represents an affinity value of the probe i. $e^{\eta ij}$ is considered as an error term.
- $\alpha_{ij} = a_j + \bar{\nu}_{ij}$, where a_j is the chip j offset and $\bar{\nu}_{ij}$ is considered as noise.

Putting everything together the model is: $y_{ij} = (a_j + \bar{\nu}_{ij}) + (\beta_j \gamma_{ij} e^{\eta ij}) x_{ij}$

Instead of trying to determine explicitly the value of the probe, we use $m_{ij} = \gamma_{ij} x_{ij}$ as a measure of the abundance of transcript. Also, we scale the additive noise, $\bar{\nu}_{ij} = \nu_{ij}/\beta_j$, so we obtain:

$$y_{kiij} = a_j + \frac{\nu_{ij}}{\beta_j} + \beta_j e^{\eta ij} m_{ij}$$

$$\frac{y_{ij} - a_j}{\beta_j} = e^{\eta ij} m_{ij} + \nu_{ij}$$

The left hand side describes the calibration of the microarray intensities Y_{ij} through subtraction of the offset a_j and scaling by normalization factor β_j.

The next step is to estimate the values of a_j y β_j so the variance is stable. For a family of random variables Y_u with expectation values $E(Y_u) = u$

and variances $Var(Y_u) = v(u)$ then $Var(h(Y_u)) \approx h'(u)^2 v(u)$, and we can do a variance-stabilizing transformation finding a function $h(y)$ so $h'(u)^2 = \frac{1}{v(u)}$.

In this case, the function $h(y)$ is the following: $h_j(y) = arcsinh\frac{y-a_j}{b_j}$

And then, after transformation:

$arcsinh\frac{y-a_j}{b_j} = \mu_{ij} + \epsilon_{ij}$

Where μ_{ij} represents the expression level after normalization.

Finally, values of a_j and β_j are estimated using a maximum-likelihood estimation.

4.3 Summarization

In the microarrays we have many probes for measuring the expression of each gene. We need, a method to summarize, a method to obtain a single expression level value from the measured probe intensity levels.

```
#Methods: "median.polish", "tukey.biweight" among others
threestep(AffyBatchObject,
        summary.method="median.polish"
)

#or also
ExpressionSet<- expresso(AffyBatchObject,
                bgcorrect.method="rma",
                normalize.method="quantiles",
                pmcorrect.method="pmonly",
                summary.method="medianpolish")
```

4.3.1 Tukey Bi-Wight

Tukey Biweight is a summarization algorithm which use the median [8], instead of mean, not to be affected by outliers. algorithm proceeds as follows:

1. For each element i the distance u_i to the center is calculated: $u_i = \frac{x_i - M}{0.5 * S + \epsilon}$ Where M is the median of the probes, S is the median of $|x_i - M|$ and ϵ is a small value to avoid division by 0.

2. The weights are calculated, depending on the distance:

$$w(u) = \begin{cases} (1+u^2)^2 & |u| \leqslant 1 \\ 0 & |u| > 1 \end{cases}$$

To exclude the outliers we set their weights to 0

3. Finally, we compute a weighted mean: $T_{bi} = \frac{\sum_{i=1}^{n} w(u)x_i}{\sum_{i=1}^{n} w(u)}$

this value T_{bi} will be used as the expression value.

4.3.2 Medianpolish

It was observed that the variability between different probes may be higher than the variability of a probe across different arrays [11]. We want to find a summarization method which also can borrow information from multiple arrays.

The median polish algorithm uses probes from multiple array. This method is the summarization process of RMA, where only PM values are used [5]. We try to fit the following model:

$$log_2(y_{ij}) = \alpha_i + \mu_j + \epsilon_{ij}$$

Where μ_j is the expression value we want to use. α_i is an affinity factor of each probe i and ϵ_{ij} is an error term.

To obtain the estimation of μ_j median polish is used. This method fits a similar model:

$$y_{ij} = \alpha_i + \beta_j + \mu + \epsilon_{ij}$$

Where $(\beta_j + \mu)$ is the value μ_j (in log_2 scale) in our model.

Now, to estimate the terms α_i,β_j and μ we use median polish algorithm as follows:

1. The following matrix is created:

$$\begin{array}{ccccc|c} e_{11} & \cdots & e_{1j} & \cdots & e_{1N_A} & a_1 \\ \vdots & \cdots & e_{ij} & \cdots & \vdots & \vdots \\ e_{I_n1} & \cdots & e_{I_nj} & \cdots & e_{I_nN_A} & a_{I_n} \\ \hline b_1 & \cdots & b_j & \cdots & b_{N_A} & m \end{array}$$

   Initially $a_i = b_j = m = 0$. In the matrix, $e_{ij} = y_{ij}$, the intensities of the probes of each array j are in each column.

2. For each row i: The median M_i of this i row (from e matrix) is calculated. To each element e_{ij}, their M_i is subtracted, and added to a_i.

3. The same thing with columns, for each column j: The median M_j, of this j column is calculated. To each element e_{ij} their M_j is subtracted, and added to b_j.

4. Steps 2 and 3 are repeated until e_{ij} elements are close to 0.

5. At the end of the process the estimations are: $\hat{\mu} = m$, $\hat{\alpha} = a$, $\hat{\beta} = b$. The elements e_{ij} are the residuals.

4.4 Probe correction

For detecting cross-hybridization level, together to each Perfect Match probe (PM) there is a Mismatch probe (MM). Initially, Affymetrix subtracted MM values to they corresponding PM values, however MM do not behave as expected and it could lead to a bad adjusted probe values.

There are preprocessing techniques (like RMA) which ignore MM values. Other methods, like the one used by Affymetrix (MAS 5.0) improved the way to adjust PM values. They use an algorithm to estimate the mismatch value, the Ideal Mismatch (IM), and then subtract it to PM values [8]:. The algorithm for the probe correction is the following:

1. A proportion of PM and MM in a probe set is calculated using Tukey biweight algorithm:

   $SB = T_{bi}(log_2(\frac{PM_p}{MM_p}) : p = 1..n_s)$

2. The Ideal Mismatch (IM) is calculated depending on three cases:

   - Si $MM < PM$, then
     $IM = MM$
   - If $SB > \tau$, (by default $\tau = 0.03$) it means, even if MM value is higher than PM, in general the PM values of the probe set are higher:
     $IM = \frac{PM}{2^{(SB)}}$
     we use a scaled PM value.
   - If $SB < \tau$, it means, in general the MM values of the probe set are higher than their PM, then:
     $IM = \frac{PM}{2^a}$
     where $a = \frac{\tau}{1+\frac{\tau - SB_s}{sc_\tau}}$ (by default $sc_\tau = 10$)

3. The calculated IM values are subtracted to PM values: $PM - IM$.

Chapter 5

Quality assessment of microarray data

In this chapter we introduce how to perform quality assessment of microarrrays. The gene expression levels are obtained through a complex procedure which may have potential variations. For this reason a control of the quality of the data is needed. We may want to remove outliers, or microarrays that are not well hybridized. Also, quality control techniques are useful to decide whether the data needs more pre-processing or not.

5.1 Quality assessment plots

We present different techniques for data visualization. The plots presented here are helpful for providing an overview of the quality of data, detecting potential problematic microarrays etc. The data used for displaying the plots in this section has been obtained from `http://www.ncbi.nlm.nih.gov/geo/query/acc.cgi?acc=GSE21779` site.

5.1.1 Probe intensities: density histograms and boxplot

To examine the raw data intensities of microarrays we use histograms and boxplots. Differences between microarrays, in the shape or center of the distribution, often means that a normalization is needed [14].

```
hist(AffyBatchObject)
boxplot(AffyBatchObject)
```

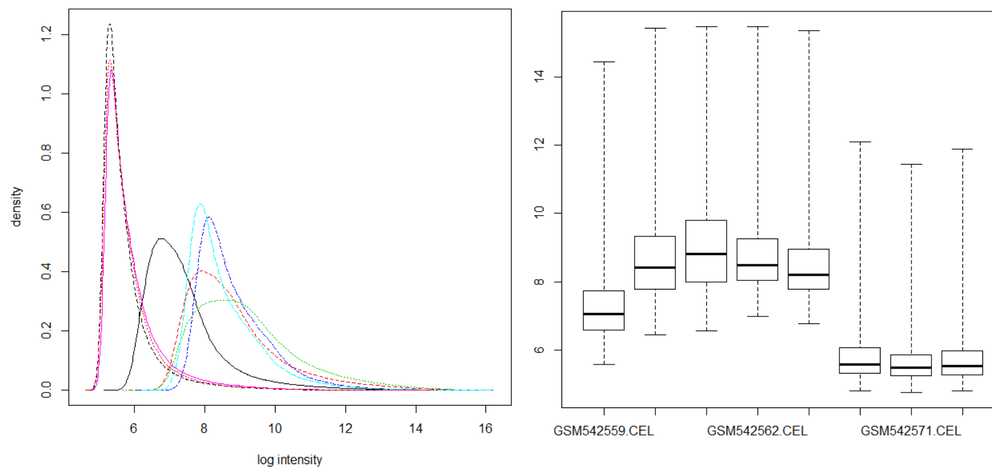


Figure 5.1: Density histogram (left) and boxplot (right)

5.1.2 QCstats

Before made comparisons between microarrays, it is necessary to check that they are of sufficient quality. Affymetrix provides a collection of quality control metrics to asses array quality [15]. The function qc , generates the most commonly used metrics:

1. *Average background*: The average background should be similar across all chips.

2. *Scale factors*: It is assumed that the majority of genes are not differentially expressed, so we expect the trimmed mean intensity of each array to be similar. However, due to non biological reasons, the intensities of arrays may not be in the same scale. Scale factor provides a measure of the overall expression level for an array. We should avoid having large differences between scale factors, Affymetrix recommend that they should be within 3-fold of one another.

3. *Number of genes called present*: Present/Marginal/Absent calls are generated looking at the diference between PM and MM. If PM values are not significantly above MM it is flagged as Marginal or Absent. Absolute value is not a good metric because it can be caused by non biological reasons. Only when we notice significant differences in % Present call between arrays they should be treated with caution.

4. *3' to 5' ratios*: Affymetrix chips contain separate probe sets targeting the 5, mid and 3 regions of Housekeeping genes (genes which are expressed at

constant levels in every cell) such as GAPDH and β-actin. We can obtain a measure of the quality of RNA comparing the amount of signal from the 3 probeset to 5 probesets. It is expected not have very large differences.

The function qc takes an AffyBatch object (it has to be unnormalized) and stats are calculated from MAS 5.0 algorithm. Those stats are used to check that arrays have been hybridised correctly and that sample quality is acceptable.

```
library (simpleaffy)
#Execute qc function
QCstats = qc(AffyBatchObject)

#Obtain the average, min, and max backgrounds
avbg(QCstats)
minbg(QCstats)
maxbg(QCstats)

#Obtain scale factors
sfs(QCstats)

#Obtain percent present
percent.present(QCstats)

#Obtain 3'/5' and 3'/mid ratios of
#actin and GAPDH
ratios(QCstats)


#Display a plot of qc stats
plot(QCstats)
```

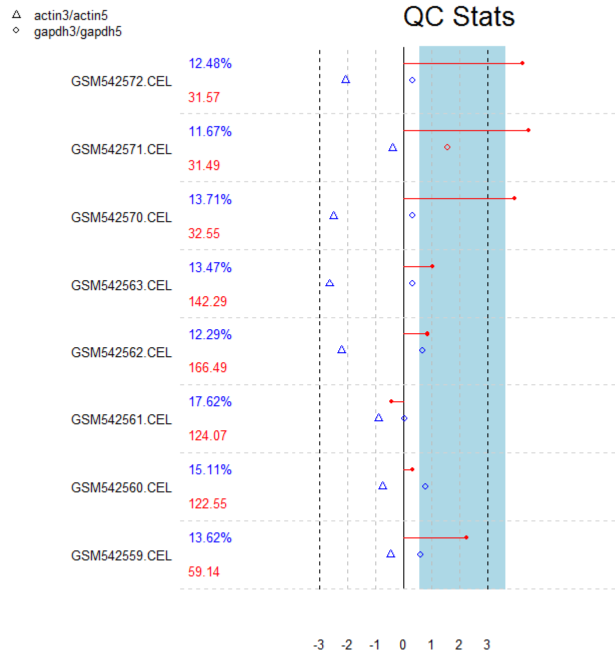


Figure 5.2: Plot of qc stats. The red dots indicates that data may need to be preprocessed

In the Figure 5.2 we have the following:

- Next to the name of the array (on the left) we have the the *percent present* (the percent above) and *average background level* (the number below). If they show high variation they will be coloured red, otherwise, blue.
- The blue strips represents the range where scale factors are within 3-fold. The dots are the *scale factor* of each array. If every dot is inside the blue strip the dots will be blue, otherwise all dots will be red.
- GAPDH 3/5 values are plotted as circles and β-actin, 3/5 ratios are plotted as triangles. If GAPDH values are considered potential outliers (when they are above 1.25) they are coloured red, otherwise they are blue. For β-actin it is recommended the ratio to be below 3. If they are, they will be coloured blue, and those above 3, red [15].

5.1.3 RLE and NUSE plots

Relative Log Expression (RLE)plot compares the expression levels of each chip to a calculated median value of all arrays [12, 13]. To display the box plot [14]:

1. Compute the log scale estimates $\hat{\theta}_{gj}$ for each gen g and each array j.

2. Compute the median m_g value across arrays for each gene.
3. Define relative expression as $M_{gi} = \hat{\theta}_{gj} - m_g$ and display a boxplot for each array j.

We assume that most genes have similar expression values, so, every array should be centred in 0 and have a similar interquartile range. Deviating boxplots often indicate problematic chips.

Normalized Unscaled Standard Error (NUSE) plot visualizes the standard error estimates for each gene and each array and standardized across arrays so the median standard error is 1 for every array[12, 13, 14].

$$NUSE(\hat{\theta}_{gj}) = \frac{SE(\hat{\theta}_{gj})}{med_i(SE(\hat{\theta}_{gj}))}$$

Low quality arrays are those that are significantly different than the other arrays.

```
library("affyPLM")
AffyBatchPLM= fitPLM(AffyBatchObject)

#RLE plot
Mbox(AffyBatchPLM, main="RLE")
abline(h=0)

#NUSE plot
boxplot(AffyBatchPLM, main="NUSE")
abline(h=1)
```

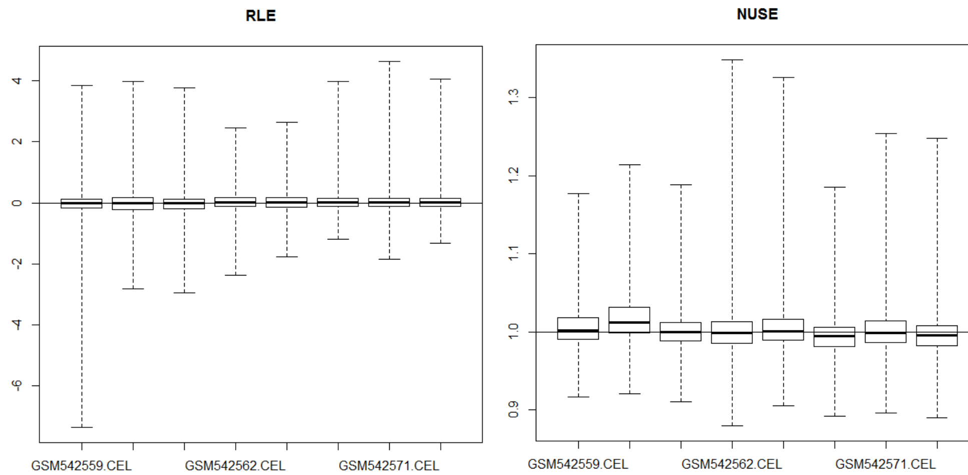


Figure 5.3: RLE plot (left) and NUSE plot (right). The fact that every box is centred on the horizontal line and are similar to each other indicates that there are no problematic array

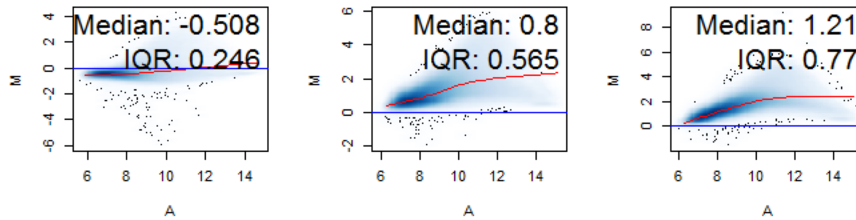
5.1.4 MA plots

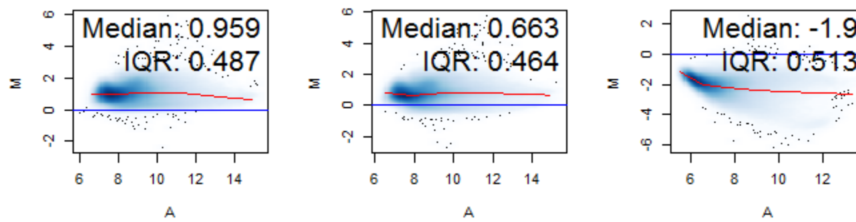
To compare two intensities I_1, I_2 instead of $I_1 = I_2$ plot, usually is plotted a 45 degrees rotated version. The rotation helps to detect patterns as deviations from horizontal, instead of diagonal. This is called MA plot [13, 14]. On the y axis we have log intensities difference (M), and on x axis the mean of their logarithms (A):

- $M = log(I_1) - log(I_2)$
- $A = [log(I_1) + log(I_2)]/2$

We expect the mass of the distribution of MA plot to be concentrated along the M = 0 axis.

```
#Display the MA-plot against a pseudo-median.
par(mfrow = c(3, 3)) #Displays 9 plots
MAplot(AffyBatchObject, plot.method = "smoothScatter")
```





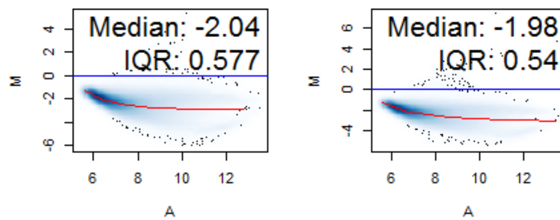


Figure 5.4: A MAplot of each microarray. Since microarrays are one-channel, each one is plotted against a pseudo-median reference chip

The curve all MAplots in Figure 5.4 should be around M=0 axis. As we can see, they are not, this means that we have to normalize.

5.1.5 RNA degradation

RNA degradation starts from the 5 end, so we expect probe intensities to be higher at 3' end and lower at 5' end. If RNA is too degraded it will have a high slope from 5' to 3'. RNA degradation plots tell us as if there are big differences in RNA degradation between arrays. The slope itself is not important, what we want is microarrays to have a similar slope [14].

```
degradeObject <- AffyRNAdeg(AffyBatchObject)
plotAffyRNAdeg(degradeObject)
```

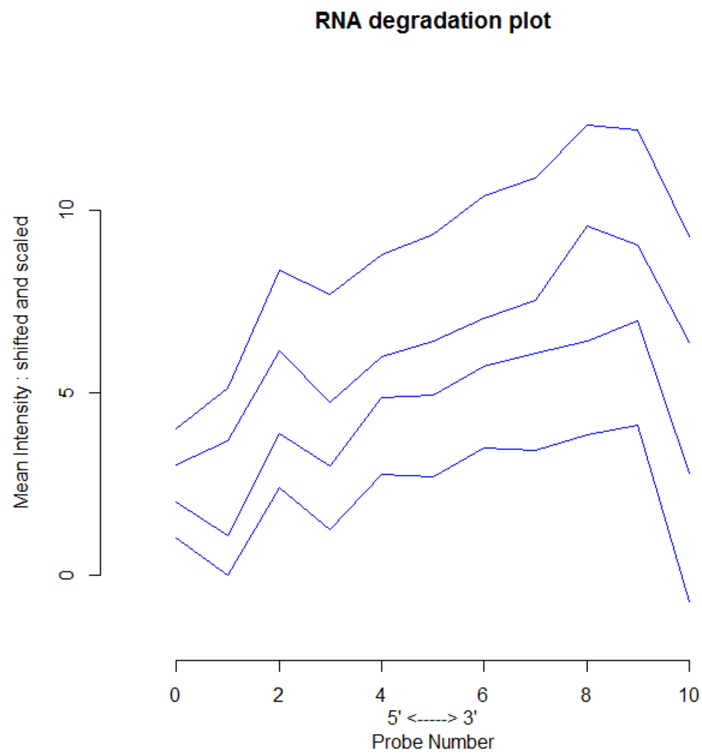


Figure 5.5: A RNA degradation plot of some microarrays

Chapter 6

Methods for filtering genes from an ExpressionSet

For doing data analysis, we are interested in the genes that are differentially expressed. For this reason feature selection algorithms are useful because we can remove genes which are similarly expressed among every sample and does not show large difference between control group and disease group.

6.1 Filter genes by variance

In this section we introduce a technique that does not need to separate samples in groups. We perform a feature selection without taking into consideration the sample annotation.

6.1.1 Non-Specific filter

A probe that has similar expression level in every sample is not very informative. It means that is similarly expressed in disease samples and control samples. We want to remove these probes and keep the ones which show large variation. As a preliminary filter we remove genes with low variation. To do that, we can use `nsFilter` function [16]. In the following example we remove the 60% of genes with lowest variance:

```
library("genefilter")

#Remove 60% of genes with lowest variance
esetFilter<-nsFilter(ExpressionSet,
                         remove.dupEntrez=FALSE,
                         require.entrez=FALSE,
                         var.cutoff=0.6)
```

```
#The object esetFilter has:
#--The log (filter.log) with the number of features removed and
#--The ExpressionSet filtered (eset)

Filterlog<-esetFilter$filter.log
ExpressionSet<-esetFilter$eset
```

6.2 Filter genes by means of parametric statistical tests

In this section we introduce a different approach of gene filtering. We use sample annotation to form two groups of samples: Control and Disease. Then, we can perform different hypothesis test between groups for each gene, The aim is to find if the expression levels of the groups are similar or not.

6.2.1 Mean difference t-test

The most simple way to compare the mean of two groups is to perform a mean difference hypothesis test. This is what we do in the following code, divide sample in two groups. The results are used to do feature selection on the ExpressionSet.

```
library("genefilter")
RowTestTable = rowttests(ExpressionSet, "CLASS_NAME")

#Crate filtered ExpressionSet
N=50 #Number of genes we want to keep
order=order(RowTestTable$p.value)[1:N]
features=featureNames(ExpressionSet)[order]
ExpressionSet=ExpressionSet[features,]
```

In the code, *CLASS_NAME* is the name of the class we want to use to form two groups (the values of *CLASS_NAME* must have only two possible values) of samples and compare the differences of the mean. We can obtain de names of the different classes using: `varLabels(ExpressionSet)`.

After executing the code we have a data frame with the columns indicating the statistics (`RowTestTable$statistic`), mean difference (`RowTestTable$dm`) and p-value (`RowTestTable$p.value`). We have decided to order the data frame by p-value, which represents how likely the results are if null hypothesis H_0 is true (how likely are the means of both groups to be the same). If p-value is small, we reject the hypothesis H_0, so we are looking for probes with large mean difference and small p-values. To represent the relation between P-values and

mean differences, we can do a volcano plot (like Figure 6.1).

```
plot(RowTestTable$dm,
        -log10(RowTestTable$p.value),
        xlab="mean differences (in log-ratio)",
        ylab=expression(-log[10]~(p-value)))
```

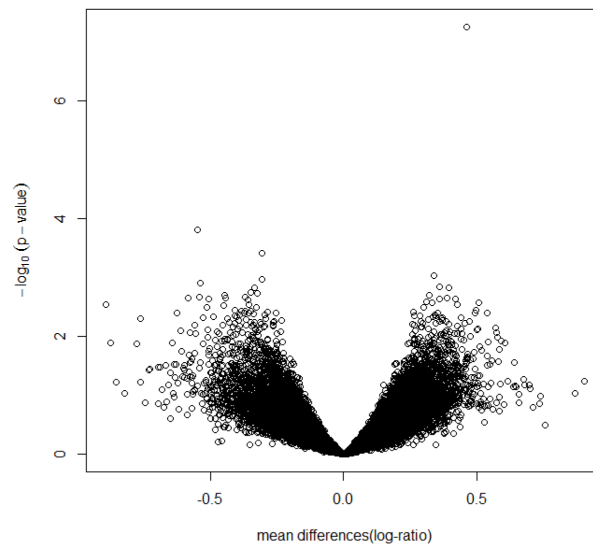


Figure 6.1: A Volcano plot. The mean difference is plotted against p-value (both in log values)

6.2.2 Multiple testing

The Westfall and Young permutation computes an adjusted p-value [20]:

1. P-values are calculated for each gene based on the original data set using Welch statistics for each gene [16].
   $t = \frac{\bar{x}_1 - \bar{x}_2}{\sqrt{\frac{s_1^2}{n_1} + \frac{s_2^2}{n_2}}}$
   The samples are divided in two groups, control and treatment. Average expression level of the groups are denoted by $\bar{x}_1$ and $\bar{x}_2$, the variance by s_1^2 and s_2^2 and the number of samples by n_1 and n_2 [21].

2. Create a pseudo-data set by dividing the data into artificial treatment and control groups.

3. P-values for all genes are computed on the pseudo-data set.

4. The successive minimum of the new p-values are retained and compared to the original ones.

5. The process is repeated a large number of times. The adjusted p-value is the proportion of resampled data sets where the minimum pseudo-p-value is less than the original p-value.

Using this procedure, the adjusted p-values obtained are usually higher than original p-values. Therefore, we may be missing a large number of differentially expressed genes, so a procedure to control the proportion of incorrectly rejected genes is needed. One of the methods is the False Discovery Rate (FDR) proposed by Benjamini and Hochberg [22]. The procedure is [20]:

1. Rank the p-value of each gene from the smallest to the largest.

2. Corrected p-value of i-th gene (from a total of n genes) is:
$corrected\ p.value = p.value * \frac{n}{i}$

3. If corrected p-value of a gene is less than 0.05 then the gene is significant.

```
library("multtest")

#Westfall and Young procedure
cl=as.numeric(CLASS_NAME==CLASS_VALUE)
resWY = mt.maxT(exprs(ExpressionSet),classlabel=cl,B=1000)

#Benjamini and Hochberg procedure
resBH = mt.rawp2adjp(resWY$rawp,proc = "BH")
```

In `resWY` object we have a data frame with a gene per row (ordered by p-value) and four columns: The index of the gene in the ExpressionSet (`resWY$index`), test statistic (`resWY$teststat`), the p-values (`resWY$rawp`) and adjusted p-values (`resWY$adjp`).

In `resBH` object we have two fields: a matrix (`resBH$adjp`) with a row for each gene and two columns: the p-value and new adjusted p-value. The other field contains the index of the genes in the ExpressionSet (`resBH$index`).

6.2.3 Moderated t-statistics

In the cases where we have few samples, instead of ordinary t-test it might be better to use moderated t-test [16]. For that, we need to fit a linear model for each gene with `lmFit` function, and then moderated t-test with `eBayes` function.

Given m sets ($y^{(i)}$,$x_1^{(i)}$, $x_2^{(i)}$ ... $x_n^{(i)}$) (with $i = 1, ..., m$) where $y^{(i)}$ is the observation and $x_1^{(i)}$, $x_2^{(i)}$ ... $x_n^{(i)}$ are the independent variables, linear model tries to find α_0,α_1...α_n so we can estimate y using the following model:

$$\hat{y}^{(i)} = \alpha_0 + \alpha_1 x_1^{(i)} + ...\alpha_n x_n^{(i)}$$

We want a different linear model to predict each gen. The following algorithm is done for each gene. In this case, each x_n is the class of the sample n (encoded as a number), and we want to estimate y, the log_2 intensity of a gene. So, we want to find α to predict the intensity given a class:
$y = \alpha * CLASS_NAME$

To make α to encode the differences between two classes we use the mean, which works as intercept α_0:
$y - mean = \alpha * CLASS_NAME$
$y = mean + \alpha * CLASS_NAME$

To calculate a linear model for each gene we can use `lmFit` function. This functions returns an object that can be used to do a moderated t-test.

The covariance of coefficients is $var(\hat{\alpha}) = Vs^2$. It is given by a multiplication of a unscaled standard deviations matrix V and residual standard deviation s^2. Both V and s^2 of all genes can be obtained from the object returned by `lmFit`, the components `fit$stdev.unscaled` and `fit$sigma`. The ordinary t-statistic would be:

$$t = \frac{\hat{\beta}}{s.e.} = \frac{\hat{\beta}}{\sqrt{var}} = \frac{\hat{\beta}}{s\sqrt{v}}$$

Where $\hat{\beta}$ is the contrast estimator and v, in the case where we have two-group comparison, is defined as $(0,1)^T V(0,1)$ to pick out the coefficient relating to the difference between the two groups [18].

In moderated t-statistic, gene-specific variance s is replaced by $\tilde{s}$, a mixture of s^2 (per-gene deviation variation) and overall estimate variation s_0^2:

$$\tilde{s}^2 = \frac{d_0 s_0^2 + ds^2}{d_0 + d} = \frac{d_0}{d_0 + d} * s_0^2 + \frac{d}{d_0 + d} * s^2$$

Where $\frac{d_0}{d_0+d}$ weight coefficient is associated with all probes and $\frac{d}{d_0+d}$ is associated with the particular gene [17, 19].

Finally, moderated t-statistic (Smyth, 2004 [17]) is the following:

$$t = \frac{\hat{\beta}}{\tilde{s}\sqrt{v}}$$

We can compute moderated t-statistics using `eBayes` function. The complete

code is:

```
library(limma)
#Create design matrix
#Convert from categorical values to numbers
cl=as.integer(ExpressionSet$CLASS_NAME)-1
design=cbind(mean=1,diff=cl)

#Do a linear model for each gene
fit=lmFit(exprs(ExpressionSet),design)
#compute moderated t-statistics
fit=eBayes(fit)
```

In the code, we first create a design matrix. The column of design matrix has, encoded as 0 or 1, the class of the n samples. In the second column each row r (with $r = 1, ..., n$) there is a 0 or 1 representing the class of the sample r. In the first column we have 1 so we obtain the mean as α_0 (every sample have the same single class, so predicted value is the mean).

The design matrix is used to calculate a linear model for each gene using `lmFit`, and then we calculate t-statistics using `eBayes`.

Chapter 7

Supervised and unsupervised learning techniques for the analysis of microarray data

The last step of the data analysis are supervised and unsupervised learning methods. Both of them are explained in this chapter.

7.1 Preparing data

Before using machine learning techniques, in this section we are going to introduce some concepts which will be useful, or even necessary, for later uses.

7.1.1 Standardization

To ensure that all genes have an equal weighting in the machine learning techniques we are going to use, a standardization of the data is needed. The standardization proposed [16] substracts row medians to each element and divide them by row IQR (Interquartile range).

```
library("matrixStats")
M<-exprs(ExpressionSet) #Obtain the matrix of ExpressionSet
esetScaled<-(M- rowMedians(M))/rowIQRs(M) #Scale ExpressionSet
exprs(ExpressionSet)<-esetScaled
```

7.1.2 Distance matrix

A distance matrix is a matrix which contains the distances between each pair of elements $(\mathbf{a}, \mathbf{b})$ from the data set. It is a $N \times N$ matrix, where N is the number of elements in the data set. Each element m_{ij} of the matrix corresponds to the distance between $\mathbf{a}$ and $\mathbf{b}$.

```
#Obtain matrix of distances
Distances=dist(exprs(ExpressionSet),method="euclidean")
```

To create a distance matrix it is necessary to define how distance are measured (a metric). The metric is set using `method` parameter. The main available distances between two points $\mathbf{a} = (a_1, ...a_c)$ and $\mathbf{b} = (b_1, ..., b_c)$ are:

- Euclidean distance (`"euclidean"`):
  $d(\mathbf{a}, \mathbf{b}) = \sqrt{\sum_{i=1}^{c}(a_i - b_i)^2}$
- Maximum distance (`"maximum"`):
  $d(\mathbf{a}, \mathbf{b}) = max\{|a_i - b_i|_i\}$
- Manhattan distance (`"manhattan"`):
  $d(\mathbf{a}, \mathbf{b}) = \sum_{i=1}^{c}|a_i - b_i|$
- Minkowski distance (`"minkowski"`):
  $d(\mathbf{a}, \mathbf{b}) = \sqrt[p]{\sum_{i=1}^{c}(a_i - b_i)^p}$

7.1.3 Principal component analysis (PCA)

Principal component analysis (or PCA) is a procedure used to reduce the number of dimensions of the data. One of the uses of PCA is for exploring data. It is possible to plot the two most informative components so we can visualize an approximate representation of the data in two dimensions.

```
PCA<-prcomp(exprs(ExpressionSet),scale.=T)
plot(PCA$x  [,1], PCA$x  [,2])

#Optionally, it is possible display gene names
text(PCA$x[,1],PCA$x[,2]-0.5,labels=featureNames(ExpressionSet))
```

7.2 Supervised learning (Classification)

Supervised learning (or classification) techniques tries to classify, to label, an element. For doing that, this techniques uses the information of data set of labelled elements.

7.2.1 KNN

KNN (K-Nearest-Neighbour) is a method of supervised classification. Given a data set of N elements $\{\mathbf{x_1}, ...\mathbf{x_N}\}$ we want to classify a new element $\mathbf{x_{N+1}}$. The class of this element will be the most common class among its k most similar elements:

1. Choose the k elements which are the closest to the new element $\mathbf{x_{N+1}}$.
2. The class of the new element is the most frequent class among selected k elements.

```
library("MLInterfaces")

fvalidation<-xvalSpec("LOO")

f<-formula(paste(CLASS_NAME, "~ ."))
Classifier = MLearn(f,
                data=ExpressionSet,
                .method=knnI(k=1),
                trainInd=fvalidation)
```

In the code the function `fvalidation` is used because it is a necessary parameter. The purpose of this function will be explained later.

7.2.2 Random forest

Random forest technique uses the data set to construct a forest of independent classification trees. Then, a new element $\mathbf{x_{N+1}}$ is classified according to the vote of the majority of trees. Given a set of N elements $\{\mathbf{x_1}, ...\mathbf{x_N}\}$, each element $\mathbf{x_i} \in \mathbb{R}^d$ multiple classification trees are constructed, each tree is created as follows [24]:

1. The root of the tree is a training set of N elements. This training set is created selecting N elements randomly with replacement.
2. Choose a subset of m elements ($m < N$) randomly from the training set.
3. Pick the best $\mathbf{x_i}$ which splits best data in two partitions for some dimension of d. Those partitions will be the children of the actual node.
4. Repeat the process (from the step 2) with each children creating a classification tree.

To classify a new element we push the element throw every classification tree. From each tree we obtain the probabilities of each class. The element is classified with the most probable class.

```
library("MLInterfaces")
fvalidation<-xvalSpec("LOO")

f<-formula(paste(CLASS_NAME, "~ ."))
Classifier =MLearn(f,
                data=ExpressionSet,
                .method=randomForestI,
                ntree=100,
                trainInd=fvalidation)
```

In this code there is an extra parameter, `ntree`, the number of trees of the forest. By default, the value of m is sqrt(number of features).

7.3 Unsupervised learning (Clustering)

Unsupervised learning (or clustering) is the process for grouping the elements of the data set into groups called clusters. The goal is having the elements which are close from one another in the same cluster, while elements from different clusters are far from one another [23]. This techniques do not use the class of the elements.In this section we introduce some of the unsupervised learning techniques.

7.3.1 K-means

Given a set of N elements $\{\mathbf{x_1}, ...\mathbf{x_N}\}$ the K-means algorithm tries to find a partition of data into k clusters. The value of k must be given by us (the plot of PCA could give us an intuition of which should it be). The k-means algorithm chooses k centroids randomly and updates them as we assign the elements of the data set [23]:

1. Initialize k cluster centroids randomly.
2. For each element $\mathbf{x_i}$, from $\mathbf{x_1}$ to $\mathbf{x_N}$:
   (a) Assign each element $\mathbf{x_i}$ to the cluster whose centroid is the closest.
   (b) Recalculate de centroids of the clusters.

```
CLUSTER_NUMBER=2
kmeans=kmeans(exprs(ExpressionSet),
        centers=CLUSTER_NUMBER,nstart=5)

#Save in a list the name of genes of each cluster
ClusterGenes<-list()
for (i in 1:CLUSTER_NUMBER){
```

```
  ClusterGenes[[i]]<-names(kmeans$cluster[kmeans$cluster==i])

}
```

7.3.2 Hierarchical clustering

Hierarchical clustering is a technique that tries to create a hierarchy of clusters. One of the advantages of this technique is that it does not require to know the number of cluster in advance, it is possible to select the most convenient partition after the hierarchy is created. Here we explain the algorithm of the agglomerative version:

1. Initially each element is a cluster so, there there are N clusters: $P_0 = \{\{\mathbf{x_1}\}, ..., \{\mathbf{x_N}\}\}$
2. Select the two closest clusters and join them
3. Go to step 2. Repeat the process until we obtain a single cluster.

```
HierarchicalClust=hclust(Distances,method="single")
plot(HierarchicalClust)
```

7.4 Evaluation

Once the classification models have been constructed, we need to estimate how good or bad they are. In this section we present techniques to evaluate supervised learning (or classification) and unsupervised learning (or clustering) algorithms.

7.4.1 Cross validation (for classification)

A method to evaluate a classification algorithm is using cross validation. This technique consist on dividing the data set in two partitions: training set and validation set. Then, the model is constructed using training set and it is evaluated with validation set. In this section we introduce two types of cross validation:

- K-fold cross validation: Data set is divided in K partitions (K folds) randomly. $K - 1$ partitions are used for training set and 1 for validation. The process is repeated k times using a different fold for validation each time.
- Leave-one-out: It is a particular case of K-fold, using K as the number of elements on the original data set.

The technique of cross validation that we want to use is set as a parameter of `MLearn`. Here we show the code of both K-fold and Leave-one-out techniques.

```
#k-fold
k=8       #there are 8 folds in this example
fvalidation<-xvalSpec("LOG", k, partitionFunc=balKfold.xvspec(k))

#Leave-One-Out
fvalidation<-xvalSpec("LOO")
```

Once the validation is done, we can construct a confusion matrix to analyse the results. Confusion matrix is a table where the number of elements which have been correctly and incorrectly classified are displayed. The following table shows the structure of a confusion matrix. We assume there are only two classes, Positive and Negative, and each element can only be assigned to one of them.

		Predicted class	
		Positive	Negative
Actual Class	Positive	True Positive (TP)	False Negative (FN)
	Negative	False Positive (FP)	True Negative (TN)

The code in R to display the confusion matrix is the following:

```
confuMat(Classifier)
```

Using confusion matrix, we can construct a metric to evaluate the classification model. Some of the most used metrics are [25]:

- Precision: The percentage of positive predictions that are correct.
  $\frac{TP}{TP+FP}$

- Recall / Sensitivity: The percentage of positive labeled instances that were predicted as positive.
  $\frac{TP}{TP+FN}$

- Specificity: The percentage of negative labeled instances that were predicted as negative.
  $\frac{TN}{TN+FP}$

- Accuracy: The percentage of predictions that are correct.
  $\frac{TP+TN}{TP+TN+FP+FN}$

7.4.2 Silhouette (for clustering)

Silhouette is a method for validating how well data has been clustered. For each element x_i a value $s(i)$ is defined to represent how well the element is clustered. $s(i)$ is calculate as follows:

$s(i) = \frac{a(i)-b(i)}{max\{a(i),b(i)\}}$

Where:

- $a(i)$ is the average distance between element x_i and the elements of the same cluster C_A (being $x_i \in C_A$):
  $a(i) = \frac{1}{n_A-1} \sum_{j \in C_A} d(i,j)$
- $b(i)$ is the distance of element x_i to the closest cluster C (being $C_A \neq C$, so $x_i \notin C$):
  $b(i) = min_{C \neq C_A} d(x_i, C)$
  The distance between an element x_i and a cluster C is defined as the average of distances between x_i and every element x_j of cluster C ($x_j \in C$):
  $d(x_i, C) = \frac{1}{n_C} \sum_{j \in C} d(x_i, x_j)$.

If $s(i)$ is large (higher than 0, close to 1), we can say that element $s(i)$ is well clustered. However, if $s(i)$ is closer to -1 it means that element x_i should be in another cluster [26]. To obtain a measure of how properly the entire data has been clustered we can calculate $\bar{s}$, the average of $s(i)$.
$\bar{s} = \frac{1}{N} \sum_{i=1}^{N} s(i)$

```
silhouette=silhouette(kmeans$cluster, Distances)
plot(silhouette)

#Obtain the average silhouette of the data
summary(silhouette)[["avg.width"]]
```

Chapter 8

A Case study on microarray data analysis for Alzheimer disease

In this chapter we explain how to perform an analysis of DNA microarrays from the raw data. For that we use the techniques explained in the previous chapters. The whole script can be found in the appendix or in my GitHub site `https://github.com/alberto-poncelas/bioc-alzheimer`

8.1 The data set

The data set used for this example is the one on the experiment of *'Microarray analyses of laser-captured hippocampus reveal distinct gray and white matter signatures associated with incipient Alzheimers disease'* [27] where some parts of the brain have been analysed for Alzheimer disease investigation. The data of microarrays of the experiment is stored in Gene Expression Omnibus with the name *Various stages of Alzheimer's disease: laser-captured hippocampal CA1 gray matter* and Reference Serie GSE28146.

The data consists of 30 samples. However for this example we only consider 15 of them: 8 of control and 7 of severe stage of Alzheimer disease.

For performing the analysis the script is prepared to load the CEL files from a folder called *data*. Therefore, so after downloading the raw data we have to place it in that folder. Also, the information about the samples needs to be load from a text file called *phenodata.txt*. This be copied from GEO DataSet browser.

8.2 Import data

First of all we are installing all packages needed. This code uses the function `installPackageList` created by us to automatically load all necessary libraries.

```
#install all required libraries
source("installPackageList.R")
libraries<-c("affy","AnnotationDbi","affyPLM","simpleaffy",
        "genefilter","matrixStats",
        "MLInterfaces","hgu133plus2.db")
installPackageList(libraries)
```

Once the CEL files have been correctly downloaded and placed in the proper folder, we need to import them into an AffyBatch object.

First, we define the folders with the main script and the folder where CEL files are stored. In the code `obtainRawData` is a helper function (the code is included in the appendix) created to automatically download raw data and put CEL files in the proper folder.

```
current_directory=getwd()

data_folder_name="data"
dataset_directory=paste(getwd(),data_folder_name,sep="/")


library("affy")
library("AnnotationDbi")


#Download raw data
source("obtainRawData.R")
obtainRawData("GSE28146",folder=data_folder_name)
```

Once the data has been downloaded we need to import files into an AffyBatch object. We can do it with `ReadAffy` function. However, we are only interested in samples which are "control" or "sever stage of Alzheimer". In this code we load the data of the samples from a text file (which later will become the sample annotation) and filter the names of microarrays by "severe stage" and "control" so we load only those files.

```
#Load phenodata (from "phenodata.txt") and name columns
phdataset<- read.table("phenodata.txt", sep="\t")
colnames(phdataset)<-c("sampleNames","age","diseaseStage","title")
```

```
#Filter control and severe indexes
#(the ones starting by "sever" or "control" in "diseaseStage" )
severeIndex<-
        grep("^severe",as.character(phdataset[,"diseaseStage"]))
ControlIndex<-
        grep("^control",as.character(phdataset[,"diseaseStage"]))

#Obtain the names of samples that are "severe" or "control"
dnames<-phdataset[c(severeIndex,ControlIndex),"sampleNames"]
dnames<-paste(dnames, ".CEL",sep = "")


#Load AffyBatch (load only .CEL of "severe" and "control" )
setwd(dataset_directory)
AffyBatchObject = ReadAffy(filenames=dnames)
setwd(current_directory)
```

The created AffyBatchObject has no information about the samples except the names, so we need to attach the information of each sample (we are specially interested in *diseaseStage*). In the following code we bind the columns of the existing sample annotation (the name of the samples) and the columns loaded in the "phenodata.txt" file. We want to keep all of the columns so we can check afterwards that the sample annotations has been correctly assigned to each sample.

```
#Attach the phenodata
#(the original phenodata with sample names
# plus the phenodata of "phenodata.txt")
pData(AffyBatchObject)<-cbind(
                        pData(AffyBatchObject),
                        phdataset[c(severeIndex,ControlIndex),])

#Check the data has been correctly assigned to each sample
pData(AffyBatchObject)
```

Now, in the AffyBatchObject all information about raw data that we need is stored. The next step is to preprocess the data we have.

8.3 Preprocess data

Now that data has been imported, the first thing we need to do is to assure that there are no potentially problematic samples. A way to check that is to display

a RLE and NUSE plot.

```
##### Check for microarrays potentially problematic ####


library("affyPLM")
AffyBatchPLM= fitPLM(AffyBatchObject)

#RLE plot
Mbox(AffyBatchPLM, main="RLE")
abline(h=0)

#NUSE plot
boxplot(AffyBatchPLM, main="NUSE")
abline(h=1)
```

In the Figure 8.1 we have the results of RLE and NUSE plots. As we can see, the box of every array are similar to each other. There is no box which differs much from the others. So we can conclude that there are no problematic samples.

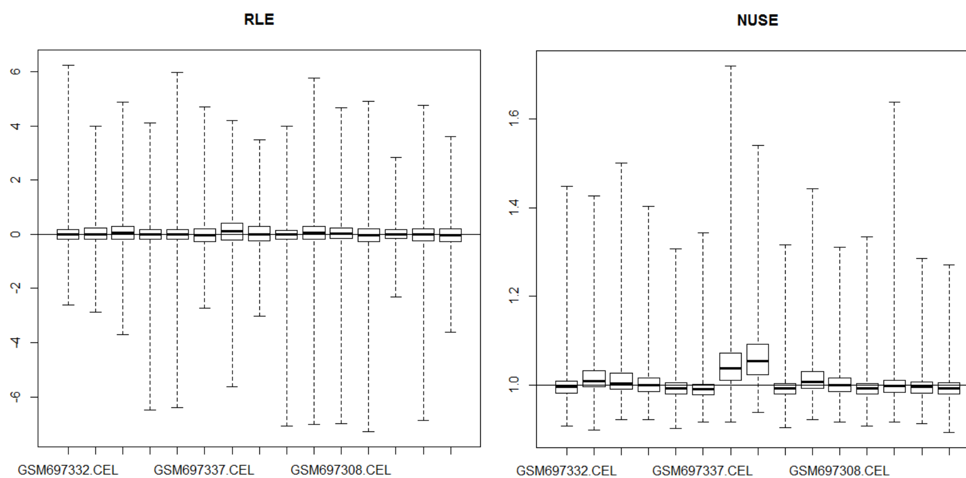


Figure 8.1: Plot of RLE and NUSE to check that there are no sample potentially problematic

Also, data needs to be normalized. If we plot an histogram of the data executing the code below (we can see the result in the Figure 8.2). Every

sample are different from each other.

```
###Check why normalization is needed###
hist(AffyBatchObject)
```

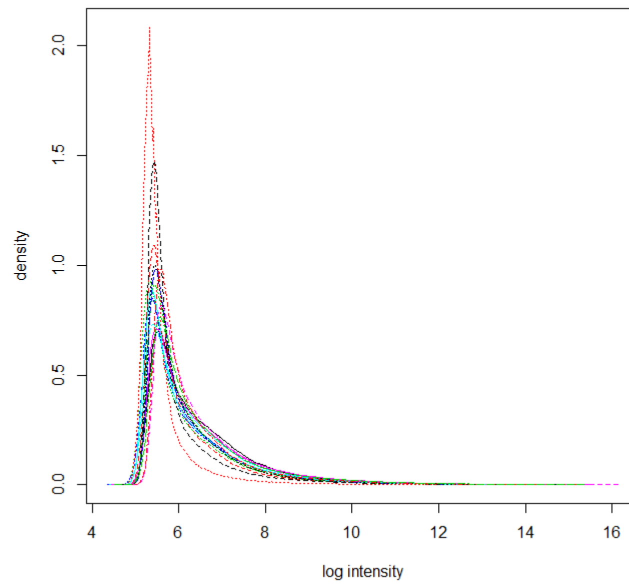


Figure 8.2: Plot of the histogram before preprocessing

To normalize data we are executing both preprocess sets explained in this document (MAS and RMA) and we will compare the results.

```
######Execute both preprocess and compare####

library("affyPLM")

##MAS preprocess
AffyBatchMAS<-preprocess(AffyBatchObject,
        background.method="MAS",
        normalize.method="scaling"
)


##RMA  preprocess
AffyBatchRMA<-preprocess(AffyBatchObject,
```

```
        background.method="RMA.2",
        normalize.method="quantile"
)

par( mfrow = c( 1, 2 ) )
hist(AffyBatchMAS)
title("Histogram with MAS")

hist(AffyBatchRMA)
title("Histogram with RMA")
```

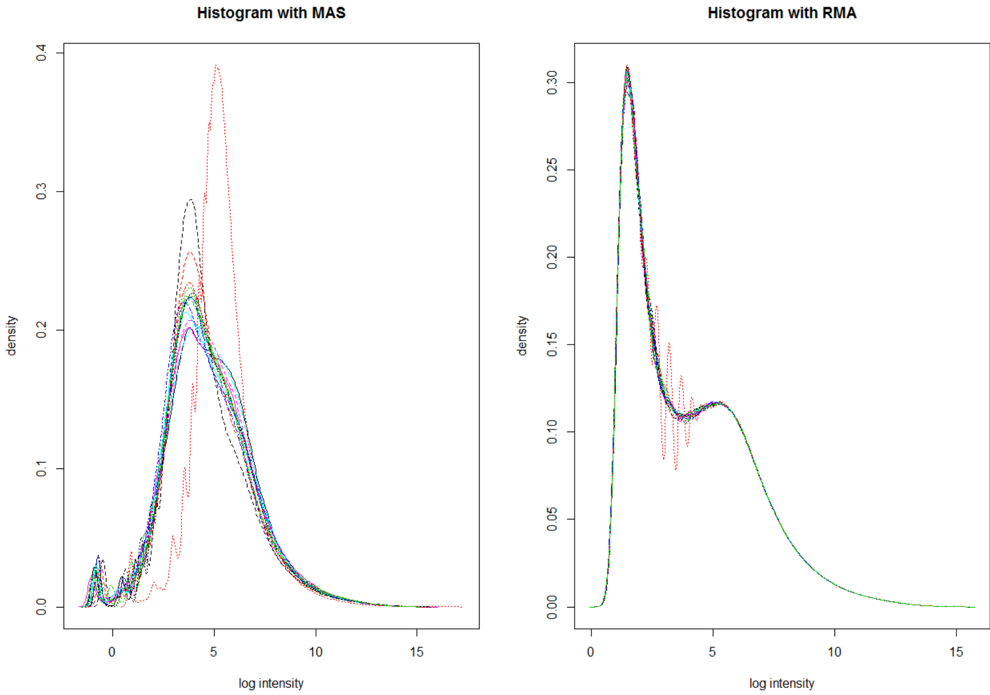


Figure 8.3: Comparation of histograms with MAS and RMA preprocessing methods

In the Figure 8.3 we have plotted both histograms so we can compare them. As we can see RMA histogram seems to be better normalized. For this reason we decide to apply RMA preprocessing technique to obtain the ExpressionSet object. To do that, we can use `expresso` function specifying every technique or `rma` function which is faster.

```
#######Convert to ExpressionSet####

ExpressionSet<- expresso(AffyBatchObject,
                bgcorrect.method="rma",
                normalize.method="quantiles",
                pmcorrect.method="pmonly",
                summary.method="medianpolish")

###Also it is possible to execute RMA directly:
### ExpressionSet= rma(AffyBatchObject)
```

8.4 Filter genes

Once we have obtained the ExpressionSet, we have everything ready to perform a data analysis. We start filtering and removing the genes that show no variation among samples, and therefore they are not involved in the disease. The following code is used for removing the 60% of the genes which shows lowest variation.

```
####Filter by variance####
library("genefilter")
ExpressionSet<-nsFilter(ExpressionSet,var.cutoff=0.6)$eset
```

We can go further and create a list with the genes whose mean expression level in "control" and "severe stage" samples which differ the most. In the following code we use he function `rowttests` to perform a mean difference hypothesis test (between "control" and "severe stage" samples).

```
####Filter by mean tests####
N=50 #Number of genes we want to keep. Keep 50 genes

varLabels(ExpressionSet)
#Two groups will be created depending on diseaseStage

# Perform a mean difference hypothesis test
# between two groups for each gene
RowTestTable = rowttests(ExpressionSet, "diseaseStage")
```

We display the `RowTestTable` in a volcano plot.

```

# Obtain the genes with the lowest p-value
order=order(RowTestTable$p.value)[1:N]
features=featureNames(ExpressionSet)[order]
```

```
# Display a plot volcano.
# We highlight with a blue dot
# the 50 points with the lowest pvalue
plot(RowTestTable$dm,
        -log10(RowTestTable$p.value),
        xlab="mean differences (in log-ratio)",
        ylab=expression(-log[10]~(p-value)))


points(        RowTestTable$dm[order],
                -log10(RowTestTable$p.value)[order],
                pch=18, col="blue")
```

The results of the volcano plot can be seen in Figure 8.4. We have highlighted the 50 genes with the lowest p-value with blue dots.

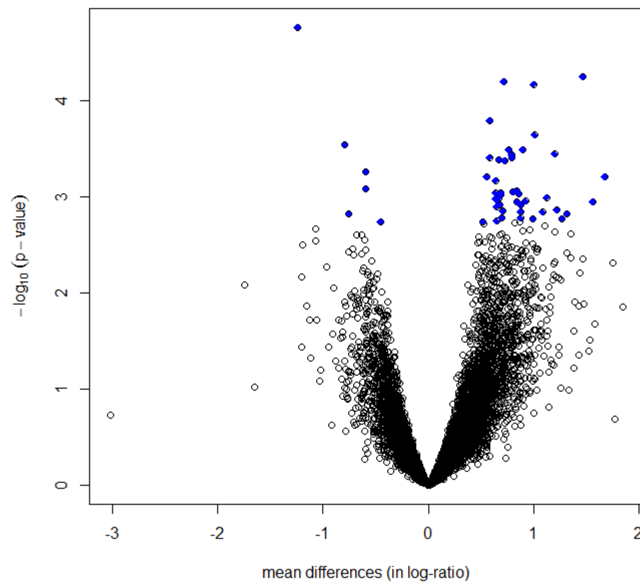


Figure 8.4: Volcano plot of the data. The 50 genes with lowest p-values are highlighted with blue dots

As final step, we order the `RowTestTable` by p-value and use them to filter genes once again.

```
#Order RowTestTable by p-value
RowTestTable<-RowTestTable[order(RowTestTable$p.value),]


# Filter the ExpressionSet and keep only obtained
# N genes (50 in this case)
ExpressionSet=ExpressionSet[features,]
```

the 50 genes of the table obtained by `rowttests` are kept in the ExpressionSet. If we execute `head(RowTestTable)` we can display the first lines of the table.

```
> head(RowTestTable)
            statistic         dm      p.value
202018_s_at -6.606320 -1.2417850 1.698699e-05
205352_at    5.855922  1.4681507 5.630073e-05
204287_at    5.788955  0.7122636 6.289332e-05
208032_s_at  5.751013  0.9984660 6.698413e-05
224311_s_at  5.244928  0.5840541 1.582479e-04
205230_at    5.041252  1.0086608 2.259021e-04
```

We can observe the first 6 rows with the lowest p-value obtained from test hypothesis. In the `dm` column there is the mean difference between two groups.

8.5 Classification

Now, in the ExpressionSet, there are only the most informative genes. Using this reduced ExpressionSet it would be easier to apply some supervised learning techniques to construct a model so we can predict if a new sample could be Alzheimer sample or not.

First of all, we start by scaling the ExpressionSet, executing the following code.

```
library("matrixStats")
M<-exprs(ExpressionSet) #Obtain the matrix of ExpressionSet
esetScaled<-(M- rowMedians(M))/rowIQRs(M) #Scale ExpressionSet
exprs(ExpressionSet)<-esetScaled
```

Once the ExpressionSet has been scaled we are going to use classification techniques. In the code below we construct a KNN to classify samples in two groups "control" and "severe stage". Also, we use Leave-One-Out to validate our model.

```
library("MLInterfaces")

fvalidation<-xvalSpec("LOO")
f<-formula(paste("diseaseStage", "~ .")) #"diseaseStage ~ ."

##### KNN ####
ClassifierKNN = MLearn(f,
                data=ExpressionSet,
                .method=knnI(k=1),
                trainInd=fvalidation)


# Obtain confusion matrix
confusionMatrixKNN<-confuMat(ClassifierKNN )
```

As well as KNN we can construct a classifier using Random Forest. In the code below we construct the model.

```
### Random forest ####
ClassifierRForest =MLearn(f,
                data=ExpressionSet,
                .method=randomForestI,
                ntree=100,
                trainInd=fvalidation)


# Obtain confusion matrix
confusionMatrixRForest<-confuMat(ClassifierRForest)
```

To estimate how good or bad our models are, or just to be able to compare them, we use a confusion matrix. We are interested only in ”control” and severe stage group” so we display only those two classes in the matrix.

```
#### Display confusion matrices###
confusionMatrixRForest<-
        confusionMatrixRForest[c("control","severe stage"),
                               c("control","severe stage")]
confusionMatrixRForest



## Display both confusion matrices
confusionMatrixKNN
confusionMatrixRForest
```

The values of confusion matrices obtained are:

```
> confusionMatrixKNN
             control severe stage
control            8            0
severe stage       1            6


> confusionMatrixRForest
             control severe stage
control            7            1
severe stage       1            6
```

The accuracy of these confusion matrices are 0.93 and 0.86. As we can see both of them give good results.

8.6 Clustering

In this section we are applying clustering techniques. First of all, we compute distance matrices for later uses. We compute two distance matrix, one with distances between samples and another one with distances between probes.

```
#####Compute distance matrix #####
# between samples, and also between probes
DistanceSamples=dist(t(exprs(ExpressionSet)),method="euclidean")
DistanceProbes=dist(exprs(ExpressionSet),method="euclidean")
```

We start performing the analysis of clustering using samples distance matrix. We first do a PCA so we can visualize the distances between samples, and then, we perform a hierarchical clustering.

```
######Clustering of samples#############
#We display PCA to have an intuition of
#the number of clusters of samples
PCAsamples<-prcomp(t(exprs(ExpressionSet)))
plot(PCAsamples$x [,1], PCAsamples$x [,2])


HierarchicalClustSamples=hclust(DistanceSamples,method="single")
plot(HierarchicalClustSamples)")
```

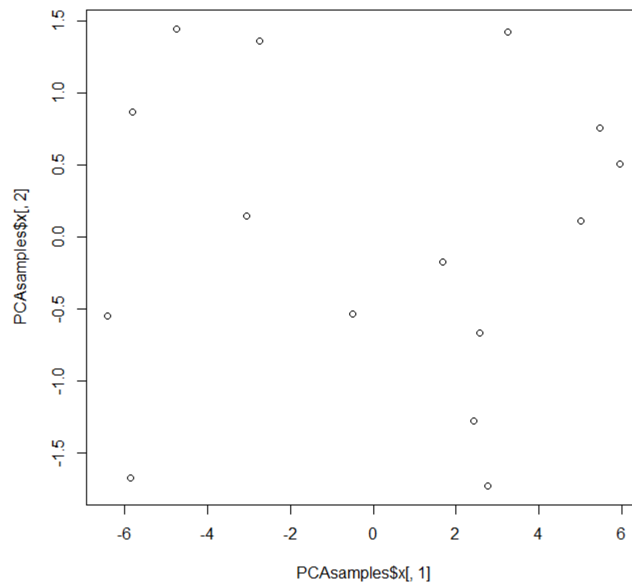


Figure 8.5: PCA of the samples

In the Figure 8.5 we show the plot of the PCA of the samples. At first sight we cannot distinguish any groups of samples. For this reason we perform a hierarchical clustering (so we do not need to decide the number of clusters) obtaining as results the Figure 8.6.

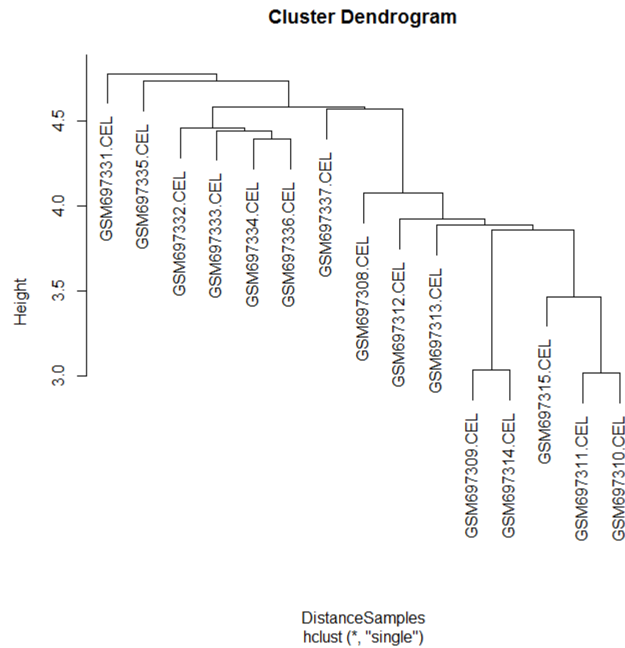


Figure 8.6: The results of the hierarchical clustering of the samples displayed as a dendrogram

Now, we are using the same techniques but in this case with distances between probes (instead of between samples).

```
###### Clustering of probes #############
#We display PCA to have an intuition of the number of clusters
PCAprobes<-prcomp(exprs(ExpressionSet))
plot(PCAprobes$x [,1], PCAprobes$x [,2])

text(PCAprobes$x[,1],
        PCAprobes$x[,2]-0.2,
        labels=featureNames(ExpressionSet))


## Hierarchical clustering of probes ####
HierarchicalClustProbes=hclust(DistanceProbes,method="single")
plot(HierarchicalClustProbes)
```

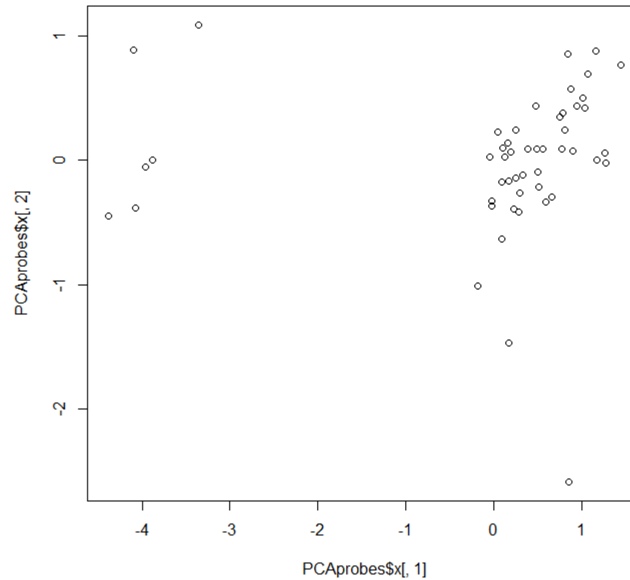


Figure 8.7: PCA of the probes showing that there are two clusters

In the Figure 8.7 we can see the results of PCA of the probes. Although the code also makes possible to display the name of the probes on the plot, we choose to plot just the points to help the visualization. This time we can observe that there are two groups. We decide then, to perform a k-means setting as 2 the number of clusters.

```
####K-means####
#We decide to set the number of clusters as 2
CLUSTER_NUMBER=2
kmeans=kmeans(exprs(ExpressionSet),centers=CLUSTER_NUMBER,nstart=5)


#Save in a list the name of genes of each cluster
ClusterGenes<-list()
for (i in 1:CLUSTER_NUMBER){
        ClusterGenes[[i]]<-names(kmeans$cluster[kmeans$cluster==i])
}
```

The second part of the code above is to create an object with two vectors, each one containing the probes of the cluster. As we can see, there is a small

cluster with only six probes, the same six probes which in Figure 8.7 are separated.

```
> ClusterGenes
[[1]]
[1] "202018_s_at" "207055_at"   "228346_at"   "229046_s_at"
[5] "211612_s_at" "223801_s_at"

[[2]]
 [1] "205352_at"    "204287_at"    "208032_s_at" "224311_s_at"
 [5] "205230_at"    "200978_at"    "213927_at"   "205635_at"
 [9] "221504_s_at"  "219145_at"    "222005_s_at" "217564_s_at"
[13] "229770_at"    "204471_at"    "228280_at"   "200703_at"
[17] "236591_at"    "203797_at"    "204141_at"   "228027_at"
[21] "235781_at"    "206045_s_at"  "205113_at"   "229818_at"
[25] "218604_at"    "214293_at"    "207507_s_at" "243521_at"
[29] "242372_s_at"  "202077_at"    "206481_s_at" "1557122_s_at"
[33] "244834_at"    "229963_at"    "226580_at"   "204229_at"
[37] "226003_at"    "209615_s_at"  "228716_at"   "213309_at"
[41] "244111_at"    "222985_at"    "225658_at"   "215045_at"
```

8.7 Obtaining information about genes

Once we have discovered the most relevant probes, we want to search which is the gene that these probes represent. We also would like to access NCBI to find more information about the genes.

First of all, we are going to create a table with probe names, their gene id and the gene name. In this case, the anotation of the *ExpressionSet* is `hgu133plus2` so we download the corresponding library and create the table.

```
annotation(ExpressionSet)

library("hgu133plus2.db")

#Entrez genes data.frame
genesDB<-merge(
                toTable(hgu133plus2ENTREZID),
                toTable(hgu133plus2GENENAME))
```

In the `genesDB` table we have the information of the probes. For this example we use the first 10 features with the smallest p-value obtained from `RowTestTable`. These probes are stored in the variable `features`.

```
# In 'features' is stored the 50 probes
# with smallest p-value (ordered)

#Obtain 10 features
features10<-features[1:10]

#Find the probes in the table
genesDB[ genesDB$probe_id %in% features10,]
```

The last code line displays the rows of the table filtered by the 10 probes we have selected, resulting:

```
> genesDB[ genesDB$probe_id %in% features10,]
         probe_id gene_id                                             gene_name
6837    200978_at    4190                      malate dehydrogenase 1, NAD (soluble)
7856  202018_s_at    4057                                       lactotransferrin
10097   204287_at    9145                                          synaptogyrin 1
11021   205230_at   22895                             rabphilin 3A homolog (mouse)
11142   205352_at    5274 serpin peptidase inhibitor, clade I (neuroserpin), member 1
11421   205635_at    8997                                    kalirin, RhoGEF kinase
12791   207055_at    9283                         G protein-coupled receptor 37 like 1
13692 208032_s_at    2892                        glutamate receptor, ionotropic, AMPA 3
19127   213927_at    4293               mitogen-activated protein kinase kinase kinase 9
28034 224311_s_at   51719                                   calcium binding protein 39
```

Now, we have the information about the genes which may be the most relevant. Looking at the table, we know the genes id numbers and the names. We can use both of them to search for more information in NCBI, `http://www.ncbi.nlm.nih.gov/gene`.

Bibliography

[1] A. Brazma, H. Parkinson, T. Schlitt, M. Shojatalab *'A quick introduction to elements of biology - cells, molecules, genes, functional genomics, microarrays'*.
http://www.ebi.ac.uk/microarray/biology_intro.html

[2] *'How Affymetrix geneChip DNA microarray work'*.
http://public.tgen.org/tgen.org/downloads/autism/Genotypingessentials.pdf

[3] Gene Expression Omnibus web site:
http://www.ncbi.nlm.nih.gov/geo/

[4] Bioconductor web site:
http://www.bioconductor.org/

[5] Freudenberg, Johannes M. "Comparison of background correction and normalization procedures for high-density oligonucleotide microarrays." Institut fur Informatik (2005): 120.

[6] Causton, Helen, John Quackenbush, and Alvis Brazma. Microarray gene expression data analysis: a beginner's guide. Wiley-Blackwell, 2009.

[7] Bar-Or, Carmiya, Henryk Czosnek, and Hinanit Koltai. "Cross-species microarray hybridizations: a developing tool for studying species diversity." TRENDS in Genetics 23.4 (2007): 200-207.

[8] Affymetrix *'Affymetrix. Statistical Algorithms Description Document.'*. Affymetrix, Inc., Santa Clara 202

[9] Bolstad, Benjamin Milo. Low-level analysis of high-density oligonucleotide array data: background, normalization and summarization. Diss. University of California, 2004.

[10] Huber, Wolfgang, et al. "Parameter estimation for the calibration and variance stabilization of microarray data." Statistical Applications in Genetics and Molecular Biology 2.1 (2003): 1008.

[11] Bolstad, Benjamin Milo. Low-level analysis of high-density oligonucleotide array data: background, normalization and summarization. Diss. University of California, 2004.

[12] Bradley, Holly Zheng. "Quality Assessment of Microarray Gene Expression Data."
http://www.ebi.ac.uk/microarray-srv/tutorials/mugen_tutorial_affy_qa.pdf

[13] G. Rustici, A. Kauffmann *'Microarray data analysis with Bioconductor'*.
http://www.ebi.ac.uk/microarray/General/Events/EMBO2009/presentations/day5/19-QM%20hands%20on.pdf

[14] R. Irizarry, R. Gentleman *'Preprocessing Affymetrix Data. Educational Materials'*.
http://www.bioconductor.org/help/course-materials/2006/biocintro_april/thurs/affy/Affy.pdf

[15] C. Wilson, S. D Pepper, C. J Miller *'QC and Affymetrix data'*.
http://bioinformatics.picr.man.ac.uk/downloads/QCandSimpleaffy.pdf

[16] Hahne, Florian, and Robert Gentleman. Bioconductor case studies. Springer, 2008.

[17] Smyth, G. K. "Statistical Applications in Genetics and Molecular bBiology." Linear models and empirical Bayes methods for assessing differential expression in microarray experiments (2004).

[18] McCarthy, Davis J., and Gordon K. Smyth. "Testing significance relative to a fold-change threshold is a TREAT." Bioinformatics 25.6 (2009): 765-771.

[19] C. Wong *'Differential Expression'*.
http://www.bioconductor.org/help/course-materials/2010/SeattleJan10/day2/DifferentialExpression.pdf

[20] Agilent Technologies, Inc. 2005 *'Multiple Testing Corrections'*.
http://www.chem.agilent.com/cag/bsp/sig/downloads/pdf/mtc.pdf

[21] Ge, Youngchao, Sandrine Dudoit, and Terence P. Speed. "Resampling-based multiple testing for microarray data analysis." Test 12.1 (2003): 1-77.

[22] Benjamini, Yoav, and Yosef Hochberg. "Controlling the false discovery rate: a practical and powerful approach to multiple testing." Journal of the Royal Statistical Society. Series B (Methodological) (1995): 289-300.

[23] Rajaraman, Anand, and Jeffrey David Ullman. Mining of massive datasets. Cambridge University Press, 2011.

[24] L. Breiman, A. Cutler *'Random Forests'*.
http://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm

[25] R. Eisner *'Basic Evaluation Measures for Classifier Performance'*.
http://webdocs.cs.ualberta.ca/~eisner/measures.html

[26] Rousseeuw, Peter J. "Silhouettes: a graphical aid to the interpretation and validation of cluster analysis." Journal of Computational and Applied Mathematics 20 (1987): 53-65.

[27] Blalock, Eric M., et al. "Microarray analyses of laser-captured hippocampus reveal distinct gray and white matter signatures associated with incipient Alzheimer's disease." Journal of Chemical Neuroanatomy 42.2 (2011): 118-126.

Appendix A

R Code

In these appendix we show the scripts used in Chapter 8. The whole script can also be found in my GitHub site https://github.com/alberto-poncelas/bioc-alzheimer

A.1 Helper functions

Here we present the code of the helper functions that have been created.

First, `obtainRawData` function, downloads .CEL files from Gene Expression Omnibus and place the .CEL files into a folder

```
# Function name: obtainRawData
# Author: Alberto Poncelas

# This function downloads .CEL files from
# Gene Expression Omnibus (see http://www.ncbi.nlm.nih.gov/sites/GDSbrowser) and
# extract the .CEL files into a folder. (for just downloading the
# compressed .tar file, use "getGEOSuppFiles" command)

# The function parameters are:
# ----accession_number: The accession number of the data set
# (the code of "series" column in http://www.ncbi.nlm.nih.gov/sites/GDSbrowser)
# ----folder_name: The name of the folder where .CEL files will be stored

# Note: This function creates and deletes temporal folders and files,
# so it is recommended to use in project's folder


obtainRawData<-function(accession_number,folder="data"){
```

```
library(GEOquery)
library(R.utils)

#Define the path of a temporal folder to extract files
temp_folder=paste(getwd(),"/temp",sep="")

#Define and create the path to save .CEL files
data_folder=paste(getwd(),folder,sep="/")
dir.create(data_folder)


#Download .tar with files from GEO
getGEOSuppFiles(GEO=accession_number)


#Obtain the path of rae data and unzip it in temporal folder
raw_file_name=paste(accession_number,"_RAW.tar",sep="")
raw_file_folder=paste(getwd(),"/",accession_number,"/",raw_file_name,sep="")
untar(raw_file_folder, exdir=temp_folder)

#Get the names of .cel.gz compressed files
cel_zip_files <- list.files(temp_folder, pattern = "cel.gz$",ignore.case=TRUE)

#Unzip the files that contains .CEL files
sapply(paste(temp_folder, cel_zip_files , sep="/"), gunzip, remove=TRUE)


#Get the names of .CEL files
cel_files <- list.files(temp_folder, pattern = ".cel$",ignore.case=TRUE)


#Copy those files from temporal folder to data folder
files_from<-paste(temp_folder, cel_files, sep="/")
files_to<-paste(data_folder, cel_files, sep="/")
file.copy(from=files_from,to=files_to)


#Delete temporal folder
unlink(temp_folder, recursive = TRUE)

#Delete downloaded .tar with raw data
raw_file_folder=paste(getwd(),"/",accession_number,sep="")
unlink(raw_file_folder, recursive = TRUE)

}
```

The `installPackageList` function takes a vector with the names of packages and install the ones which have not been installed yet.

```
# Function name: installPackageList
# Author: Alberto Poncelas

# When in a script we need to lead a library from Bioconductor
# which is not installed, we need to install it.
# This function takes a vector with the names of packages
# and install the ones which have not been installed yet

# The function parameters are:
# ----packageList: a vector with the names of the packages
#                        that need to be installed


installPackageList<-function(packageList){

        #Get installed packages
        installedPackages<-rownames(installed.packages())


        checkInstalled<-packageList %in% installedPackages

        if (! all(checkInstalled)==TRUE){
                #Get the list of uninstalled packages
                uninstalled<-packageList[checkInstalled==FALSE]

                #Prepare Bioconductor
                source("http://bioconductor.org/biocLite.R")
                old.packages(repos=biocinstallRepos())
                library("Biobase")

                #install uninstalled packages
                for (pckg in uninstalled){
                        biocLite(pckg)
                }

        }


        # load libraries
        for (lib in packageList){
                print(lib)
```

```
                do.call(library, list(lib))

        }

}
```

A.2 Main script

Here we present the complete code of the main script for microarray data analysis.

```
#install all required libraries
source("installPackageList.R")
libraries<-c("affy","AnnotationDbi","affyPLM","simpleaffy","genefilter",
        "matrixStats","MLInterfaces","hgu133plus2.db")
installPackageList(libraries)


#####################################################
############# IMPORT DATA ##########################
#####################################################


current_directory=getwd()

data_folder_name="data"
dataset_directory=paste(current_directory,data_folder_name,sep="/")


library("affy")
library("AnnotationDbi")


#Download raw data
source("obtainRawData.R")
#obtainRawData("GSE28146",folder=data_folder_name)


#Load phenodata (from "phenodata.txt") and name columns
phdataset<- read.table("phenodata.txt", sep="\t")
colnames(phdataset)<-c("sampleNames","age","diseaseStage","title")
```

```
#Filter control and severe indexes
#(those which start by "sever" or "control" in "diseaseStage" column )
severeIndex<-grep("^severe",as.character(phdataset[,"diseaseStage"]))
ControlIndex<-grep("^control",as.character(phdataset[,"diseaseStage"]))

#Obtain the names of samples that are "severe" or "control"
dnames<-phdataset[c(severeIndex,ControlIndex),"sampleNames"]
dnames<-paste(dnames, ".CEL",sep = "")



#Load AffyBatch (load only .CEL of "severe" and "control" )
setwd(dataset_directory)
AffyBatchObject = ReadAffy(filenames=dnames)
setwd(current_directory)


#Attach the phenodata
#(the original phenodata with sample names plus the phenodata of "phenodata.txt")
pData(AffyBatchObject)<-cbind(
                        pData(AffyBatchObject),
                        phdataset[c(severeIndex,ControlIndex),])

#Check the data has been correctly assigned to each sample
pData(AffyBatchObject)



#####################################################
############PREPROCESS##############################
#####################################################


##### Check for microarrays potentially problematic ####


library("affyPLM")
AffyBatchPLM= fitPLM(AffyBatchObject)

#RLE plot
Mbox(AffyBatchPLM, main="RLE")
abline(h=0)

#NUSE plot
boxplot(AffyBatchPLM, main="NUSE")
```

```
abline(h=1)


###Check why normalization is needed

hist(AffyBatchObject)

library (simpleaffy)
#Execute qc function
QCstats = qc(AffyBatchObject)
#Display a plot of qc stats
plot(QCstats)



######Execute both preprocess and compare


library("affyPLM")

##MAS preprocess
AffyBatchMAS<-preprocess(AffyBatchObject,
        background.method="MAS",
        normalize.method="scaling"
)


##RMA  preprocess
AffyBatchRMA<-preprocess(AffyBatchObject,
        background.method="RMA.2",
        normalize.method="quantile"
)

par( mfrow = c( 1, 2 ) )
hist(AffyBatchMAS)
title("Histogram with MAS")

hist(AffyBatchRMA)
title("Histogram with RMA")


#######Convert to ExpressionSet

ExpressionSet<- expresso(AffyBatchObject,
                bgcorrect.method="rma",
```

```
                normalize.method="quantiles",
                pmcorrect.method="pmonly",
                summary.method="medianpolish")

###Also it is possible to execute RMA directly:
### ExpressionSet= rma(AffyBatchObject)


############################################################
##################### FILTER GENES #######################
############################################################

####Filter by variance####
library("genefilter")
ExpressionSet<-nsFilter(ExpressionSet,var.cutoff=0.6)$eset


####Filter by mean tests####
N=50 #Number of genes we want to keep. Keep 50 genes

varLabels(ExpressionSet)
#Two groups will be created depending on diseaseStage

# Perform a mean difference hypothesis test
# between two groups for each gene
RowTestTable = rowttests(ExpressionSet, "diseaseStage")

# Obtain the genes with the lowest p-value
order=order(RowTestTable$p.value)[1:N]
features=featureNames(ExpressionSet)[order]


# Display a plot volcano.
# We highlight with a blue dot
# the 50 points with the lowest pvalue
plot(RowTestTable$dm,
        -log10(RowTestTable$p.value),
        xlab="mean differences (in log-ratio)",
        ylab=expression(-log[10]~(p-value)))


points(        RowTestTable$dm[order],
                -log10(RowTestTable$p.value)[order],
                pch=18, col="blue")
```

```
#Order RowTestTable by p-value
RowTestTable<-RowTestTable[order(RowTestTable$p.value),]


# Filter the ExpressionSet and keep only obtained
# N genes (50 in this case)
ExpressionSet=ExpressionSet[features,]



###########################################################
################     CLASSIFICATION    ####################
###########################################################

#####Scale ExpressionSet###
library("matrixStats")
M<-exprs(ExpressionSet) #Obtain the matrix of ExpressionSet
esetScaled<-(M- rowMedians(M))/rowIQRs(M) #Scale the ExpressionSet
exprs(ExpressionSet)<-esetScaled




library("MLInterfaces")

fvalidation<-xvalSpec("LOO")
f<-formula(paste("diseaseStage", "~ .")) #f is "diseaseStage ~ ."

##### KNN ####
ClassifierKNN = MLearn(f,
                data=ExpressionSet,
                .method=knnI(k=1),
                trainInd=fvalidation)


# Obtain confusion matrix
confusionMatrixKNN<-confuMat(ClassifierKNN )





### Random forest ####
ClassifierRForest =MLearn(f,
```

```
                data=ExpressionSet,
                .method=randomForestI,
                ntree=100,
                trainInd=fvalidation)


# Obtain confusion matrix
confusionMatrixRForest<-confuMat(ClassifierRForest)


#### Display confusion matrices###
confusionMatrixRForest<-
                confusionMatrixRForest[c("control","severe stage"),
                                                c("control","severe stage")]
confusionMatrixRForest



## Display both confusion matrices
confusionMatrixKNN
confusionMatrixRForest







############################################################
##################     CLUSTERING    #######################
############################################################



#####Compute distance matrix #####
# between samples, and also between probes
DistanceSamples=dist(t(exprs(ExpressionSet)),method="euclidean")
DistanceProbes=dist(exprs(ExpressionSet),method="euclidean")




######Clustering of samples############
#We display PCA to have an intuition of
#the number of clusters of samples
PCAsamples<-prcomp(t(exprs(ExpressionSet)))
```

```
plot(PCAsamples$x [,1], PCAsamples$x [,2])


HierarchicalClustSamples=hclust(DistanceSamples,method="single")
plot(HierarchicalClustSamples)



######Clustering of probes#################
#We display PCA to have an intuition of the number of clusters
PCAprobes<-prcomp(exprs(ExpressionSet))
plot(PCAprobes$x [,1], PCAprobes$x [,2])

text(PCAprobes$x[,1],
        PCAprobes$x[,2]-0.2,
        labels=featureNames(ExpressionSet))




##Hierarchical clustering of probes####
HierarchicalClustProbes=hclust(DistanceProbes,method="single")
plot(HierarchicalClustProbes)



####K-means####
#We decide to set the number of clusters as 2
CLUSTER_NUMBER=2
kmeans=kmeans(exprs(ExpressionSet),centers=CLUSTER_NUMBER,nstart=5)

#Save in a list the name of genes of each cluster
ClusterGenes<-list()
for (i in 1:CLUSTER_NUMBER){
        ClusterGenes[[i]]<-names(kmeans$cluster[kmeans$cluster==i])
}


ClusterGenes
```

```
###########################################################
#############     GENE INFORMATION    ######################
###########################################################



annotation(ExpressionSet)

library("hgu133plus2.db")

#Entrez genes data.frame
genesDB<-merge(
               toTable(hgu133plus2ENTREZID),
               toTable(hgu133plus2GENENAME))





# In 'features' is stored the 50 probes
# with smallest p-value (ordered)

#Obtain 10 features
features10<-features[1:10]

#Find the probes in the table
genesDB[ genesDB$probe_id %in% features10,]
```