An Introduction to Machine Learning with Probabilities, in R (2021-2022)

Sylvain Rubenthaler

Contents

Preface v

Chapter 1. Introduction 1
1.1. Various type of machine learning 1
1.1.1. Predictive or supervised learning 1
1.1.2. Descriptive or unsupervised learning. 2
1.2. Supervised learning 2
The need for probabilistic predictions. 2
Real-world applications 5
1.3. Unsupervised learning 5
1.3.1. Discovering clusters 5
Real-world application: 6
1.3.2. Discovering latent factors 6
Real-world application 6
1.4. Some important concepts in machine learning 10
1.4.1. A single non-parametric classifier: the K-nearest neighbor (KNN) 10
1.4.2. The curse of dimensionality 11
1.4.3. Parametric models for classification and regression 11
1.4.4. Over-fitting and under-fitting 14
1.4.5. Model selection 15
1.5. Examples in R 16
1.5.1. Iris dataset 16
1.5.2. Pima Indian diabetes dataset 18
1.5.3. Summary of useful R commands 18

Chapter 2. Linear regression 19
2.1. Introduction 19
2.1.1. Generalities 19
2.1.2. Housing values 19
2.1.3. Relationship between the input data and the selling price (continuation of the example above) 19
2.1.4. Fitting the model 20
2.2. Model specification 20
2.3. Maximum likelihood estimator 20
2.3.1. Derivation of the MLE (σ^2 known) 22
2.3.2. Geometric interpretation 23
2.4. Ridge regression 24
2.5. The LASSO (Least Absolute Shrinkage and Selection Operator) 25
2.6. Advantages/disadvantages of linear regression 25
Advantages 25
Disadvantages 25
2.7. Examples in R 26
2.7.1. BigMart data set. 26
2.7.2. Boston Housing data set 28
2.7.3. Summary of useful R commands 28

2.8. Dummy variables 28
2.8.1. Introduction to dummy variables 28
2.8.2. Exercise with dummy variables 29

Chapter 3. Logistic regression 31
3.1. Introduction 31
3.1.1. Outline of the plan 31
3.1.2. Credit card fraud detection 31
3.2. Mathematical model 32
3.2.1. Maximum Likelihood Estimator (MLE) 32
3.2.2. Optimization algorithms (non-exhaustive list) 35
3.3. l_2-regularization (for logistic regression) 37
3.4. Advantages/disadvantage 37
Advantages 37
Disadvantages 37
3.5. Examples 37
3.5.1. Detecting credit card fraudulent activity (continuation of Section 3.1.2) 37
3.5.2. Credit card default (activity) 38
3.5.3. Summary of useful R commands 39

Chapter 4. Support Vector Machine (SVM) 41
4.1. Introduction 41
4.2. Linear support vector machine (SVM) 41
4.2.1. The margin 41
4.2.2. Maximizing the margin 42
4.2.3. Support vectors 45
4.2.4. Soft margins 45
4.2.5. Advantages/disadvantages 45
Advantages: 46
Disadvantages: 46
4.3. Kernelized SVM 46
4.3.1. Description 46
4.3.2. Most popular kernels 47
4.3.3. Tuning γ 48
4.3.4. Kernelization 48
4.3.5. Advantages/disadvantages of the kernelized SVM 48
4.4. Examples in R 48
4.4.1. Toy example for linear SVM 48
4.4.2. Exercise on a toy example (using the commands we have learned above) 49
4.4.3. Toy example for kernelized SVM 50
4.4.4. Car acquisition 50
4.4.5. Digit recognition (activity) 52
4.4.6. Summary of useful R commands 53

Chapter 5. Bayes classifier and naive Bayes classifier. 55
5.1. The Bayes classifier 55
5.1.1. Introduction 55
5.1.2. Likelihood (probability of the kind $\mathbb{P}(X^{(\text{new})} = x^{(\text{new})} | Y^{(\text{new})} = c, \mathscr{D})$) (or class-conditional distribution). 55
5.1.3. Prior class distribution ($\mathbb{P}(Y^{(\text{new})} = c|\mathscr{D})$). 56
5.2. Naive Bayes classifier (NBC). 56
5.2.1. Introduction 56
5.2.2. Maximum Likelihood Estimator for the Naive Bayes Classifier 57
5.2.3. Newsgroup example (continued) 57
5.2.4. Bayesian NBC 58

5.3. Advantages/disadvantages of NBC 58
Advantages 58
Disadvantages 58
5.4. Examples in R 59
5.4.1. The naiveBayes command 59
5.4.2. Iris dataset 59
5.4.3. Naïve Bayes on SMS data 60
5.4.4. Movies reviews classification (activity) 62
5.4.5. Summary of useful R commands 62

Chapter 6. Neural networks 65
6.1. Introduction 65
6.1.1. Foreword 65
6.1.2. Biological neurons 65
6.1.3. Model neurons in a neural network. 66
6.2. Structure of the network 66
6.3. Fitting neural networks 67
6.4. Some issues in training neural networks 69
6.4.1. Starting values 69
6.4.2. Over-fitting 69
6.4.3. Scaling of the inputs 69
6.4.4. Number of hidden units and hidden layers 70
6.4.5. Multiple mining 70
6.5. Advantages/disadvantages of neural networks 70
Advantages 70
Disadvantages 70
6.6. Examples in R 70
6.6.1. Dividends 70
6.6.2. Bank marketing data set 75
6.6.3. House price (activity) 77
6.6.4. Summary of useful R commands 77

Chapter 7. Tree based methods 79
7.1. Decision trees 79
7.1.1. Regression trees 79
7.1.2. Classification trees 82
7.1.3. Comparison with linear models 84
7.1.4. Advantages/disadvantages of trees 86
7.2. Bagging, random forests 86
7.2.1. Bagging 86
7.2.2. Random forests 88
7.3. Examples in R 88
7.3.1. Baseball (Hitters dataset) 88
7.3.2. Boston Housing dataset (same as in Chapter 2) 91
7.3.3. Titanic dataset 94
7.3.4. In-vehicle coupon recommendation data set ([**WRDV**$^+$**17**]) 96
7.3.5. Useful commands in R 97

Chapter 8. Markov chain Monte-Carlo inference 99
8.1. Introduction with an example 99
8.2. Gibbs sampler 100
8.2.1. Theoretical description 100
8.2.2. Ising model 100
8.2.3. Bayesian image analysis 103
8.3. Metropolis-Hastings algorithm 104

8.3.1. Description of the algorithm 104
8.3.2. Gibbs sampling is a special case of MH (MH=Metropolis-Hastings) 105
8.3.3. Why MH works 105
8.3.4. Deciphering example 106
8.4. Advantages/disadvantages of MCMC methods 108
Advantages: 108
Disadvantages: 108
8.5. Examples in R 108
8.5.1. Volleyball 108
8.5.2. Pareto model 114
8.5.3. World Cup problem ([**AH20**]) (activity) 119
8.5.4. Summary of useful R commands 120

Chapter 9. Clustering 121
9.1. K-means clustering 121
9.1.1. Choosing the number of clusters K 123
9.1.2. Where K-means fails 123
9.1.3. Kernelized K-means. 124
9.1.4. Advantages/disadvantages of K-means methods 124
9.2. Hierarchical clustering 125
9.2.1. Advantages/disadvantages of agglomerative clustering 126
9.3. Density-based spatial clustering of applications with noise (DBSCAN) 127
9.3.1. Definitions 127
9.3.2. Algorithm 127
9.3.3. Advantages/disadvantages of DBSCAN 128
9.4. Examples in R 128
9.4.1. Mall Customer Segmentation 128
9.4.2. Cheese classification (hierarchical clustering) 134
9.4.3. Seeds dataset (activity) ([**CNK$^+$10**]) 135
9.4.4. Useful commands in R 136

Chapter 10. Appendix 139
10.1. Lagrange multipliers 139
10.1.1. Notations 139
10.1.2. Optimization under equality constrains 139
10.1.3. Optimization under inequality constrains 140

Bibliography 143

List of symbols 145

Index 147

Preface

Hello. Welcome to this course on machine learning. My name is Sylvain Rubenthaler. I am "maître de conférences" at the university of Nice Sophia Antipolis. This job title is similar to the U.S. "associate professor". During the course, we will do math exercises and build examples on computer. We will use the language R, a user-friendly language, rich in machine learning related commands.
To write this handout, I used various books and articles (cited in the references), as well as numerous websites such as: https://towardsdatascience.com, https://stackoverflow.com, https://www.edureka.co, https://www.geeksforgeeks.org, https://uc-r.github.io, www.kaggle.com, https://rpubs.com, https://www.datacamp.com, https://en.wikipedia.org, https://machinelearningmastery.com, http://archive.ics.uci.edu, http://www.stat.umn.edu, http://www2.stat.duke.edu, https://medium.com, http://mlwiki.org, https://eric.univ-lyon2.fr, http://www.math.u-bordeaux.fr/~mchave100p/, ...

CHAPTER 1

Introduction

We will explain here what is machine learning and present some applications of machine learning.

1.1. Various type of machine learning

We present here the two main types of algorithms one can encounter in the machine learning field. We do not talk about reinforcement learning because it is a different story altogether. We will talk about MCMC methods (Markov Chain Monte-Carlo) in Chapter 8 because these methods are ubiquitous in the field.

1.1.1. Predictive or supervised learning. We have an input object x and we want to predict an output y. As the computer will make a prediction on the picture, we call this setting: "predictive learning". The computer cannot do this from scratch. We have access to a training set named $\mathscr{D}$. This set is made of various inputs labelled with the correct outputs: $\mathscr{D} = \{(x_i, y_i)\}_{1 \leq i \leq N}$.

EXAMPLE 1.1. We have pictures of cats and pictures of dogs . We want to present pictures to the machine and we want it to say wether it sees a cat or a dog (see Figure 1.1.1). We have $N = 1000$ pictures

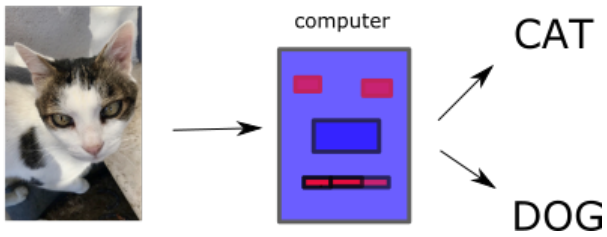


FIGURE 1.1.1. Classification task

with labels attached to the pictures (see Figure 1.1.2). This is our training set.

FIGURE 1.1.2. Labelled pictures

The computer should learn from the training set then, when a new input comes by, he will predict its label (in the example: CAT or DOG). As the computer learns from what we give it, we call this setting: "supervised learning".

1.1.2. Descriptive or unsupervised learning. Here, we have only inputs: $\mathscr{D} = (x_i)_{1\leq i\leq N}$. The goal is to find "interesting patterns in the data". This is not a well defined problem. That makes it difficult and interesting.

EXAMPLE 1.2. We have pictures of cats and dogs. We would like the computer to detect that the set of pictures is made of two different populations. Suppose we represent the data as points in $\mathbb{R}^d$ (see Figure 1.1.3) The answer of the machine could be: "I have drawn a boundary between two populations

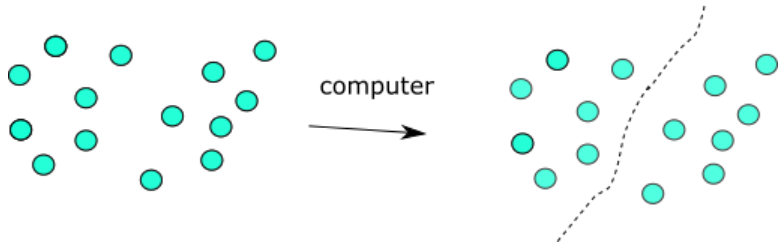


FIGURE 1.1.3. Finding a boundary

which have different characteristics.". It is then up to the user to realize he has pictures of cats on one side and pictures of dogs on the other side.

1.2. Supervised learning

We are now going with more details what is supervised learning.

Suppose the outputs belong to a set $\{1,2,\dots,C\}$ (C being the number of classes). The goal is to learn a mapping from input x to output y. If $C = 2$, this is called binary classification. If $C \geq 3$. this is called multi-class classification. Assume for each input x, the correct output y is equal to $f(x)$ for some function f. If we get to learn what is f then we can predict the output for any input.

EXAMPLE 1.3. We want to put label 0 and 1 on objects. Table 1.2.1 is our training data set. Here

Color	Shape	Size	Label
Blue	Square	10	1
Red	Ellipse	2.4	1
Red	Ellipse	20.7	0

TABLE 1.2.1. Classifying objects

comes: a blue crescent (size 5), a yellow circle (size 8), a blue arrow (size 10.5) We are required to generalize what we know from the training set. We put the label 1 on the blue crescent because all the blue shapes a labelled 1. This is weak but sounds right. What should we do for the yellow circle and the blue arrow? The correct answer is not clear. This is maybe because giving a definite answer is not the best way to tackle the problem.

The need for probabilistic predictions. Suppose we have a model : the inputs x behave in a random way, and so do the outputs y, plus we are able to compute probabilities. We compute $\mathbb{P}(y = c|x,\mathscr{D})$ (x is the input and $\mathscr{D}$ is the training set). We can then give our best guess:

$$\hat{y} = \underset{1\leq c\leq C}{\arg\max}\,\mathbb{P}(y = c|x,\mathscr{D})\,.$$

Remember that the notation argmax means that we are looking for some c maximizing the probability $\mathbb{P}(y = c|x,\mathscr{D})$. Keep in mind that it is sometimes better to say "I do not know", for example if all the $\mathbb{P}(y = c|x,\mathscr{D})$ are equally small, or if you are in a risk-averse setting (as medicine or finance).

EXERCISE 1.1. We suppose the height of male students follows a law $\mathcal{N}(\mu_M,\sigma_M^2)$ and the height of female students follows a law $\mathcal{N}(\mu_F,\sigma_F^2)$[1]. We have a training set with input=height and output=genre:

$$\mathscr{D} = \{(x_1, M), \dots, (x_n, M), (x_{n+1}, F), \dots, (x_{n+N}, F)\},$$

the first n students in $\mathscr{D}$ are male, and the last N are female. We know we have consistent estimators of μ_M, σ_M^2, μ_F, σ_F^2:

$$\frac{x_1 + x_2 + \cdots + x_n}{n} \xrightarrow[n\to+\infty]{\text{a.s.}} \mu_M,$$
$$\frac{1}{n}\sum_{i=1}^{n}\left(x_i - \frac{(x_1 + \cdots + x_n)}{n}\right)^2 \xrightarrow[n\to+\infty]{\text{a.s.}} \sigma_M^2,$$
$$\frac{x_{n+1} + x_2 + \cdots + x_{n+N}}{N} \xrightarrow[N\to+\infty]{\text{a.s.}} \mu_F,$$
$$\frac{1}{N}\sum_{i=n+1}^{N}\left(x_i - \frac{(x_{n+1} + \cdots + x_N)}{N}\right)^2 \xrightarrow[N\to+\infty]{\text{a.s.}} \sigma_F^2,$$

where a.s. stands for "almost surely". Remember, that an estimator of μ_M is any quantity built with the data we have $(x_1, x_2, \dots)$ (we also say it is a statistics of $x_1, x_2, \dots$). This estimator of μ_M is said to be consistent if it actually converges to μ_M when the quantity of data goes to infinity (here n goes to infinity). The same is true for σ_M^2, μ_F, σ_F^2.

Question: Using what we know about the distributions and supposing that the data gives us μ_M, σ_M^2, μ_F, σ_F^2, compute the probability that a new student of size x is a boy.

Answer: Let $\delta > 0$. We first work under the assumption that the height of the new student is a random variable $X \in [x, x+\delta]$. The question does not give us the full frame we need to solve the problem. We have to make our own model . Here, we suppose that, not knowing anything, the gender of the new student is a random variable G such that: $\mathbb{P}(G = M) = \mathbb{P}(G = F) = 1/2$.

In the language of Bayesian statistics, we say that the a priori law on the gender is: 50% chances of being a girl. Now a new information comes by: the height X belongs to $A := [x, x+\delta]$. We want to compute $\mathbb{P}(G = M|X \in A)$. In Bayesian statistics, this is called the a posteriori law . So, let us compute:

$$\begin{aligned}\mathbb{P}(G = M|X \in A) &= \frac{\mathbb{P}(\{G = M\} \cap \{X \in A\})}{\mathbb{P}(X \in A)} \\ &= \frac{\mathbb{P}(X \in A|G = M) \times \mathbb{P}(G = M)}{\mathbb{P}(G = M) \times \mathbb{P}(X \in A|G = M) + \mathbb{P}(G = F) \times \mathbb{P}(X \in A|G = F)}\end{aligned}$$

(using classical formulas on conditional probabilities[2]). So

$$\mathbb{P}(G = M|X \in A) = \frac{\int_x^{x+\delta} f_M(t)\,dt \times \frac{1}{2}}{\int_x^{x+\delta} f_M(t)\,dt \times \frac{1}{2} + \int_x^{x+\delta} f_F(t)\,dt \times \frac{1}{2}},$$

where f_M is the density of $\mathcal{N}(\mu_M, \sigma_M^2)$ ($f_M(t) = e^{-(t-\mu_M)^2/(2\sigma_M^2)}/\sqrt{2\pi\sigma_M^2}$) and f_F is the density of $\mathcal{N}(\mu_F, \sigma_F^2)$.

Now we want to compute $\mathbb{P}(G = M|X = x)$. We know that

$$\mathbb{P}(G = M|X = x) = \lim_{\delta\to 0} \mathbb{P}(G = M|X \in [x, x+\delta]).$$

As f_M and f_F are continuous, we get

$$\int_x^{x+\delta} f_M(t)\,dt \underset{\delta\to 0}{\sim} \delta f_M(x),$$
$$\int_x^{x+\delta} f_F(t)\,dt \underset{\delta\to 0}{\sim} \delta f_F(x).$$

And so

$$\mathbb{P}(G = M|X = x) = \frac{f_M(x)}{f_M(x) + f_F(x)}. \tag{1.1}$$

[1] Not very realistic because in this model, heights can be negative!

[2] For any events B, C, D: $\mathbb{P}(B|C) = \mathbb{P}(B \cap C)/\mathbb{P}(C)$ and if, in addition, $C \cap D = \emptyset$ and $\mathbb{P}(C \cup D) = 1$, then $\mathbb{P}(B) = \mathbb{P}(B|C)\mathbb{P}(C) + \mathbb{P}(B|D)\mathbb{P}(D)$.

REMARK 1.4. In the exercise above, we are able to compute, for a new student of height x, what is the probability that this student is a boy. Observe that we had to impose our model (by saying that that probability of an unknown person to be a boy is $1/2$, a sound prediction) and that the algorithm we propose to compute this probability is fairly simple (here: use Equation (1.1)). On the Figure 1.2.1, we

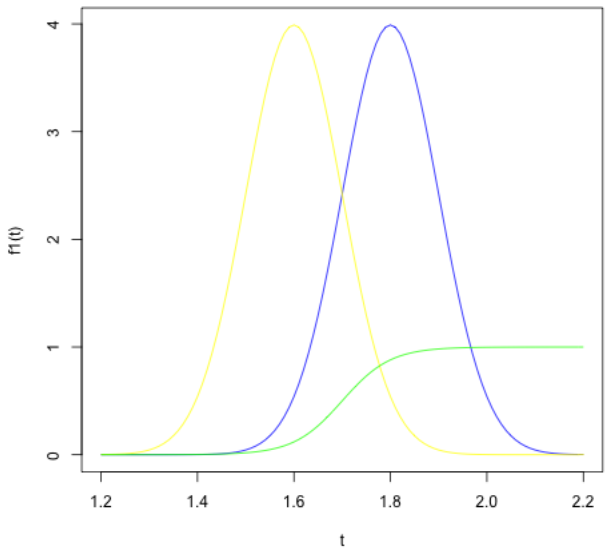


FIGURE 1.2.1. Distributions

have:

- blue curve: distribution of the male's height,
- yellow curve: distribution of the female's height,
- green: at point x, probability that a person of height x is a boy.

Here: $\mu_M = 1.8$, $\mu_F = 1.6$, $\sigma_M^2 = \sigma_F^2 = 0.01$.

EXERCISE 1.2. We are given a bag of coins in which three fourth of the coins are un-biased (when we flip a coin, the probability to get head or tail is exactly $1/2$) and the other one fourth is biased. For the biased coins, we know that each time we flip a coin, the probability to get a head is $2/3$. We sample a coin from the bag and flip it three times, we get three heads.

Question: Would you rather bet that this coin is biased or un-biased? and why?

Answer: We define the events: U ="coin is un-biased", B="coin is biased". We set X_1, X_2, X_3 to be the results of the flipping (we write $X_1 = H$, $X_2 = H$, $X_3 = H$ to say that the three results are "head"). We compute

$$\begin{aligned}
\mathbb{P}(U|X_1 = X_2 = X_3 = H) &= \frac{\mathbb{P}(U, X_1 = X_2 = X_3 = H)}{\mathbb{P}(X_1 = X_2 = X_3 = H)} \\
&= \frac{\mathbb{P}(X_1 = X_2 = X_3 = H|U)\mathbb{P}(U)}{\mathbb{P}(U)\mathbb{P}(X_1 = X_2 = X_3 = H|U) + \mathbb{P}(B)\mathbb{P}(X_1 = X_2 = X_3 = H|B)} \\
&= \frac{(1/2)^3(3/4)}{(3/4)(1/2)^3 + (1/4)(2/3)^3} \\
&\approx 0.559.
\end{aligned}$$

So we should bet that the coin is un-biased.

Real-world applications.

- Document classification (special case: spam filter).
- Image classification and handwriting recognition: indoor/outdoor, dog/cat, ... In the special case where the image consists of isolated handwritten letters and digits, we use classification to perform handwriting recognition.
- When you record a bird song into a mp3 file, it is numerically encoded. Suppose you have a data set made of bird songs coded into mp3 files, each of them with a label telling which bird species is singing. Then, when recording a new sound, an algorithm can tell you to which cluster it belongs with the highest probability, i.e. which species is the singer belonging to. Various cool apps of this kind are available for smartphones (try BirdNET for android and Bird Song Id UK for iPhone). See Figure 1.2.2 for the BirdNET app in action.

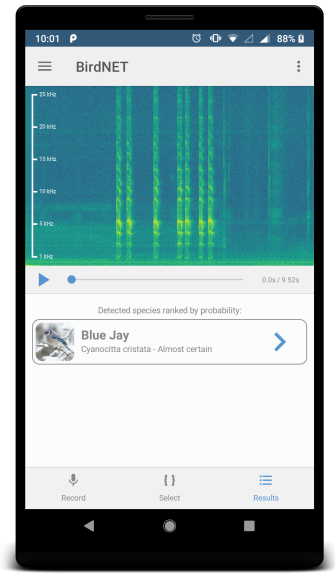


FIGURE 1.2.2. Song identification application.

1.3. Unsupervised learning

1.3.1. Discovering clusters. Here, we have a graph (Figure 1.3.1) where we have reported the weight vs. height of various individuals. We want to classify this into clusters. In this picture, we see that we

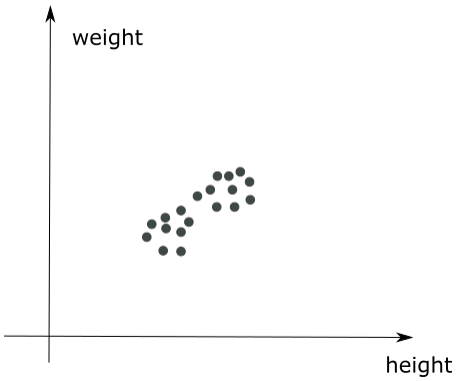


FIGURE 1.3.1. Weight vs. height

can separate the points into two clouds (corresponding to the men and women).

In general, let K be the number of clusters. Our first goal is to approximate the distribution of K, that is compute

$$\mathbb{P}(K = k|\mathscr{D})$$

for each k (assuming we have a model which allows to compute these probabilities). We can then take

$$K^* = \arg\max_k \mathbb{P}(K = k|\mathscr{D}).$$

Our second goal is to estimate which cluster each point belongs to. Let z_i in $\{1,2,\ldots,K\}$ represent the cluster to which the datapoint i is assigned. The z_i is an example of a hidden or latent variable since it is not observed in the data set. We can infer which cluster each data point belongs to by computing $z_i^* = \arg\max_k \mathbb{P}(z_i = k|x_i,\mathscr{D})$ (assuming we have a model which allows to compute these probabilities).

Real-world application:

- In e-commerce, you can cluster costumers by their purchasing and web-surfing behavior. Then you can identify the clusters (one is made of women older than fifty, and so on). And then you can send send customized targeted advertising to each group.
- In biology, the flow-cytometry is a technique used to detect and measure physical and chemical characteristics of a population of cells. You take a population of cells, you flow them one by one in a flow-cytometer. Each cell has different characteristics (the machine emits light and measure the characteristics of the light scattered by the cell). You can cluster the data points into different populations. It can help to discover different sub-populations of cells.

1.3.2. Discovering latent factors. When dealing with high-dimensional data, we can reduce the dimension by projecting the data on a lower dimensional subspace. This is called dimensionality reduction.

The motivation behind this technique is that, although the data may appear high-dimensional, there may only a small number of degrees of variability, corresponding to latent factors.

EXAMPLE 1.5. We have points in $\mathbb{R}^3$ in Figure 1.3.2. When we project the points on the axis $\{x = y = z\}$, we see two clusters. We say that the coordinate on the axis $\{x = z = y\}$ is a hidden variable.

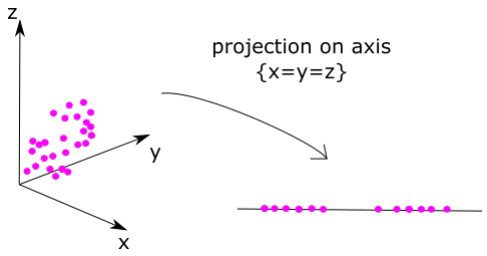


FIGURE 1.3.2. Projection

Real-world application.

- In signal processing (e.g., of acoustic or neural signals), it is common to use dimensionality reduction to separate signals into their different sources.

Introduction to Principal Component Analysis. There is an automatic way of reducing the dimension, it is called Principal Component Analysis (PCA). Let us have a look at the dataset in Figure 1.3.3 for house prices (in thousands of dollar). Each house has four features, meaning that it can be represented in $\mathbb{R}^4$. So it is difficult to visualize. Looking carefully, we see that not all features are equally important. For example, the number of floors does not help to distinguish one house from another. This feature has low variances (the numbers are almost equal). The area and value have the biggest variances so these two are more representative than the others of our data.

But there is something else to be noted. If you look at the first two features, value and area, you notice that the value is roughly double the area. So we can in fact deduce one feature from the other. This property is called covariance: the value and area feature have a high covariance, which implies

House No.	Value	Area	Floors	Household
1	148	72	4	20
2	156	76	4	22
3	160	86	4	22
4	165	79	4	24
5	169	88	5	30
6	184	90	5	35

FIGURE 1.3.3. House prices

redundancy in the data (there is more information than needed because we can deduce one feature from the other).

So, from the preceding discussion, we gather two principles.

- It is a good thing to have features with high variance, since they will be more informative and more important.
- It is a bad thing to have features with a high covariance since they can be deduced from one another, thus keeping them together is redundant.

Principal Component Analysis: the mathematics. We represent our data by N points $(y^{(1)}, y^{(2)}, \ldots, y^{(N)})$ (in $\mathbb{R}^M$). We can always make the following simplifying assumption:

$$\overline{y} = \frac{1}{N}\sum_{i=1}^{N} y^{(i)} = 0\,.$$

If this was not the case, we could subtract $\overline{y}$ to each $y^{(i)}$. We now want to create an artificial feature by forming a linear combination of all the features. For each n, $x_1^{(n)} = (w^{(1)})^T y^{(n)}$ for some vector $w^{(1)}$. Remember, this scalar product simply means: $x_1^{(n)} = \sum_{i=1}^{M} w_i^{(1)} y_i^{(n)}$. The first interesting thing we notice is

$$\overline{x}_1 = \frac{1}{N}\sum_{i=1}^{N} (w^{(1)})^T y^{(i)} = \frac{1}{N}(w^{(1)})^T \sum_{i=1}^{N} y^{(i)} = 0\,,$$

which means that the empirical mean of this artificial feature is zero. We also have

$$\begin{aligned}
\widehat{\sigma}_1 &= \frac{1}{N}\sum_{i=1}^{N} (x_1^{(i)})^2 \\
&= \frac{1}{N}\sum_{i=1}^{N} ((w^{(1)})^T y^{(i)})^2 \\
&= \frac{1}{N}\sum_{i=1}^{N} ((w^{(1)})^T y^{(i)})((y^{(i)})^T w^{(1)}) \\
&= \frac{1}{N}\sum_{i=1}^{N} (w^{(1)})^T y^{(i)} (y^{(i)})^T w^{(1)} \\
&= (w^{(1)})^T \left(\frac{1}{N}\sum_{i=1}^{N} y^{(i)}(y^{(i)})^T\right) w^{(1)}\,,
\end{aligned}$$

which means that the empirical variance of this artificial feature is

$$\widehat{\sigma}_1 = (w^{(1)})^T C w^{(1)}\,,$$

where $C = \frac{1}{N}\sum_{i=1}^{N} y^{(i)}(y^{(i)})^T$. The matrix C is real-valued and symmetric so it has a orthonormal basis, for eigenvalues $\lambda_1, \ldots, \lambda_M$ such that $|\lambda_1| \geq |\lambda_2| \geq \ldots$. We take $w^{(1)}$ such that it is equal the unit eigenvector associated to λ_1. This guarantees that the empirical variance $\widehat{\sigma}_1$ is the biggest possible

We then take, for i in $\{2,3,\ldots,M\}$, $w^{(i)}$ such that it is equal to the unit eigenvector associated to λ_i. These vectors $w^{(i)}$ are used to form artificial features. This method grantees that the so-formed features have a high variance (it decreases when i increases) and have low covariances.

Example in R. We use the data about house prices we discussed above. The following code creates the data frame (with the same shape and the same values as in Figure 1.3.3).

```
no<-c(1,2,3,4,5,6)
value<-c(148,156,160,165,169,184)
area<-c(72,76,86,79,88,90)
floors<-c(4,4,4,4,5,5)
household<-c(20,22,22,24,30,35)
house.prices<-data.frame(no,value,area,floors,household)
```

We then want to perform a PCA. As we said, we center and the data (option `center=TRUE`). We rescale the data so that the total variance is 1 (option `scale=TRUE`). And then we call for a summary.

```
house.prices.pca<-prcomp(house.prices[c(2:5)],center=TRUE,scale=TRUE)
summary(house.prices.pca)
```

The summary is the following.

```
Importance of components:
                          PC1     PC2     PC3     PC4
Standard deviation     1.8885 0.51562 0.40831 0.03197
Proportion of Variance 0.8916 0.06647 0.04168 0.00026
Cumulative Proportion  0.8916 0.95807 0.99974 1.00000
```

Let's make sense of these:

- Standard deviation: This is simply the eigenvalues in our case since the data has been centered and scaled.
- Proportion of variance: This is the amount of variance the components account for the data, i.e. PC1 accounts for > 89.1% of total variance in the data alone!
- Cumulative Proportion: This is simply the accumulated amount of explained variance, ie. if we used the first two components we would be able to account for >95% of total variance in the data.

We explain more than 95% of the variance with just the first two components. Let us plot these using

```
plot(house.prices.pca$x[,1],house.prices.pca$x[,2],xlab="PC1 (89.2%)",
ylab="PC2 (6.6%)", main="PC1 / PC2 - plot")
```

(see Figure 1.3.4 for the result). We see that there are clearly two clusters (one on the right, one on the left). Now we want to do a fancy plot where we see the original features

```
library(devtools)
install_github("vqv/ggbiplot")
library(ggbiplot)
ggbiplot(house.prices.pca)
```

(see the result in Figure 1.3.5). Here, we see the same points as in the preceding Figure, but we also draw the projections of the unit vectors of the original axis onto the subspace spanned by PC1 and PC2. As the projections of these vectors are close to one another, we can say that all the features have a high covariance with one another.

ACTIVITY. Use the `mtcars` dataset (which is built into R). This dataset consists of data on 32 models of car, taken from an American motoring magazine (1974 Motor Trend magazine). For each car, you have 11 features, expressed in varying units (US units). They are as follows.

- `mpg`: Fuel consumption (Miles per (US) gallon): more powerful and heavier cars tend to consume more fuel.
- `cyl`: Number of cylinders: more powerful cars often have more cylinders

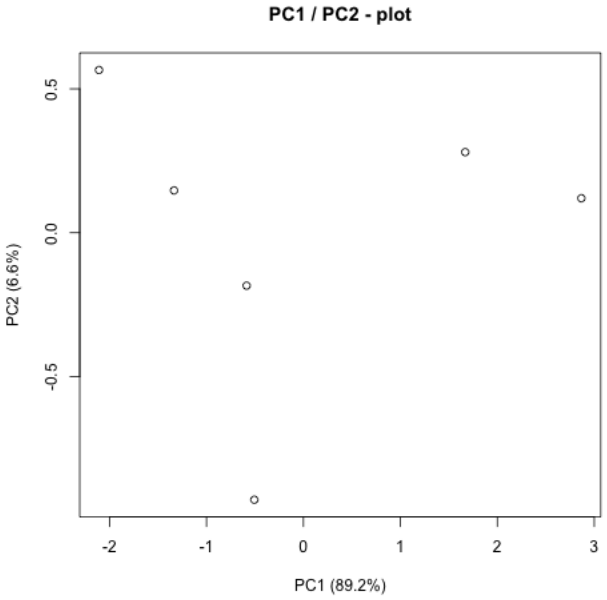


FIGURE 1.3.4. PC1 / PC2

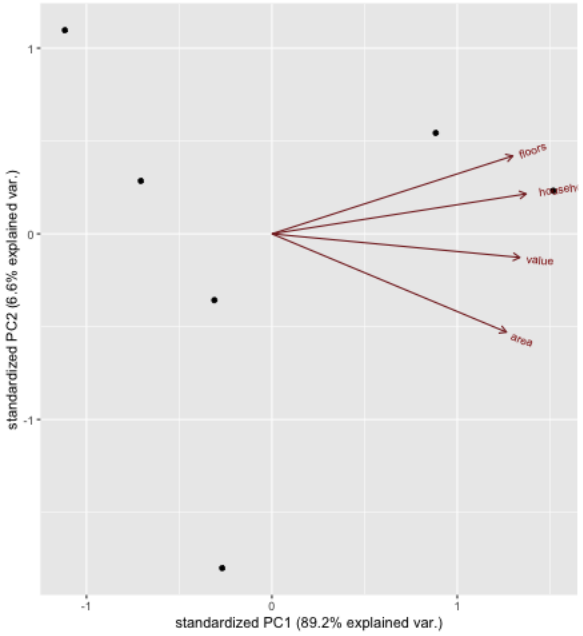


FIGURE 1.3.5. PC1 / PC2 with original feature vectors

- `disp`: Displacement (cu.in.): the combined volume of the engine's cylinders
- `hp`: Gross horsepower: this is a measure of the power generated by the car
- `drat`: Rear axle ratio: this describes how a turn of the drive shaft corresponds to a turn of the wheels. Higher values will decrease fuel efficiency.
- `wt`: Weight (1000 lbs): pretty self-explanatory!
- `qsec`: 1/4 mile time: the cars speed and acceleration

- `vs`: Engine block: this denotes whether the vehicle's engine is shaped like a "V", or is a more common straight shape.
- `am`: Transmission: this denotes whether the car's transmission is automatic (0) or manual (1).
- `gear`: Number of forward gears: sports cars tend to have more gears.
- `carb`: Number of carburetors: associated with more powerful engines

Because PCA works best with numerical data, you'll exclude the two categorical variables (`vs` and `am`). You will perform a PCA. Let is call `mtcars.pca` the result. As above, we will use `ggbiplot`, but this time with the option `labels=rownames(mtcars)` which allows to see the car names on the plot. You should see that the sports cars are clustered together (one could interpret other clusters in the same way). If you need help, see the program `chap-01-pca-activity.R`.

You can even group cars by categories (US, Japanese, European) by using

```
mtcars.country <- c(rep("Japan", 3), rep("US",4), rep("Europe", 7),
rep("US",3), "Europe", rep("Japan", 3), rep("US",4), rep("Europe", 3), "US",
rep("Europe", 3))
ggbiplot(mtcars.pca, ellipse=TRUE,  labels=rownames(mtcars), groups=mtcars.country)
```

This will get you a nice graph on which you can observe that American cars are characterized by high values for `cyl`, `disp`, and `wt`, and so on.

1.4. Some important concepts in machine learning

A model with a fixed number of parameters is a parametric model. Everything else is a non-parametric model.

1.4.1. A single non-parametric classifier: the K-nearest neighbor (KNN). We have a training set $\mathscr{D} = \{(x_i, y_i), 1 \le i \le N\}$, where each point x_i has a label y_i. The points x_i are supposed to be in $\mathbb{R}^d$, fo some d. For a new input x, we want to predict a label or the probability that the correct label is a particular y. The KNN classifier takes the input x and at the K nearest points in $\mathscr{D}$ (see Figure 1.4.1). Our probability prediction is a fraction. In the example, $K = 3$ and there are two classes: blue and pink.

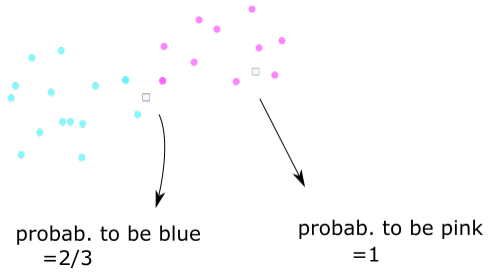


FIGURE 1.4.1. KNN

If the three nearest neighbors of x are pink, then I predict that x is pink with probability 1. If two of the neighbors are blue, I predict that x is blue with probability 2/3. And so on ...

We return the empirical fraction

$$\mathbb{P}(y = c|x, \mathscr{D}, K) = \frac{1}{K} \sum_{i \in N_K(x,\mathscr{D})} \mathbb{1}_{y_i = c}$$

(this is our prediction). The set $N_K(x, \mathscr{D})$ is the set $\{j : x_j$ is one of the K nearest neighbors$\}$. So, for each c in $\{1, 2, \ldots, C\}$, we compute the number of people in the K nearest neighbors of x who have the label c, and then divide it by K.

This is an example of memory-based learning, or instance-based learning. We usually use the euclidean distance (good if the data is real-valued) although other metrics can be used.

REMARK 1.6. The euclidean distance is isotropic, meaning the length of a stick will not vary if you rotate it. For this reason, it is important to re-scale the data (see supplementary material).

Suppose we have the following data

$$\begin{pmatrix} x_{1,1} \\ x_{2,1} \\ x_{3,1} \end{pmatrix}, \begin{pmatrix} x_{1,2} \\ x_{2,2} \\ x_{3,2} \end{pmatrix}, \ldots, \begin{pmatrix} x_{1,N} \\ x_{2,N} \\ x_{3,N} \end{pmatrix}$$

(in three dimensions). For each coordinate we compute the empirical mean and empirical standard deviation. That is, for $i = 1,2,3$,

$$\overline{x}_i = \frac{x_{1,i} + x_{2,i} + \cdots + x_{N,i}}{N}, \overline{\sigma}_i = \left(\frac{1}{N} \sum_{j=1}^{N} (x_{i,j} - \overline{x}_i)^2 \right)^{1/2}.$$

1.4.2. The curse of dimensionality. The KNN classifier performance decreases with the dimension. This is known as the curse of dimensionality. The same will be true of many algorithms relying on clouds of points to explore the space.

Suppose we have inputs which are uniformly distributed in the d-dimensional unit cube. Suppose we want to have a fraction f of the set $\mathscr{D}$ in a neighborhood of a point to be able to make a sound prediction.

To make it simpler, we use the following distance:

$$d((x_1,\ldots,x_d),(x'_1,\ldots,x'_d)) = \sup(|x_1 - x'_1|,\ldots,|x_d - x'_d|).$$

We look at a cube of size $2l$ centered in x (this is a ball of radius l for the above distance) in Figure 1.4.2. In order to get a fraction of the data into this small cube, we need l to be such that $(2l)^d/1^d = f$ (because the expectation of the number of points in the small cube is $(2l)^2 \times \#\mathscr{D}$. So we want $l = f^{1/d} := e_d(f)$. If $d = 10$ and $f = 10\%$ then $e_d(f) = 0.8$. If $d = 10$ and $f = 1\%$ then $e_d(f) = 0.63$. See also Figure 1.4.3 to have an idea of the behavior of $e_d(f)$. So, to get a decent portion f of $\mathscr{D}$, we have to look at "neighbors" which are far away ($e_d(f) = 0.63$ above), so they might not be good predictors about the input-output function in x.

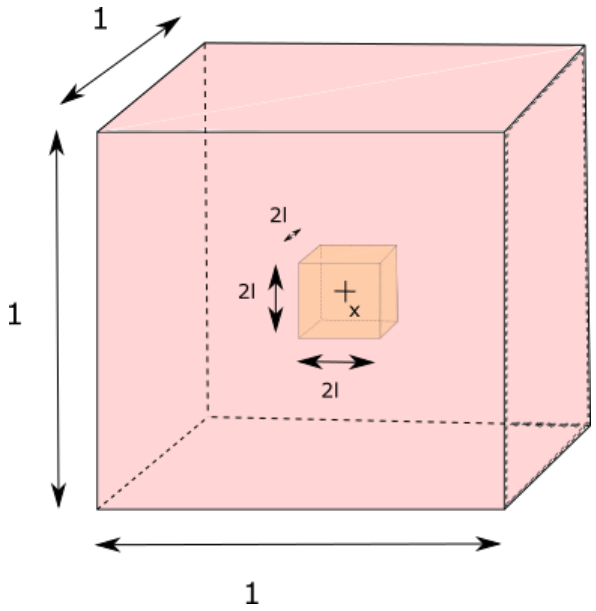


FIGURE 1.4.2. Hyper-cube

1.4.3. Parametric models for classification and regression. To fight the curse of dimensionality, we can make assumptions about the nature of the data distribution. These assumptions often take the form of a parametric model. We will now describe two examples.

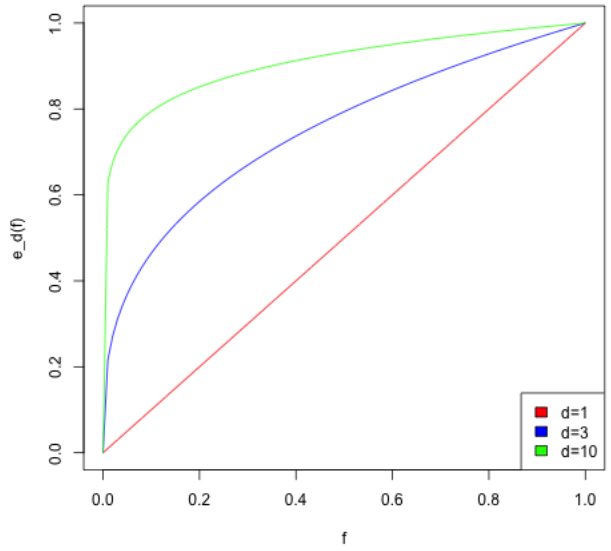


FIGURE 1.4.3. Dimensionality curse

Linear regression . We suppose the output is a linear function of the input. More precisely:

$$y(x) = w^T x + \epsilon = \sum_{i=1}^{d} w_i x_i + \epsilon\,,$$

where w, x are to be understood as column vector, ϵ is a random noise (d is the dimension). Linear regression can be made to model non-linear relationship by replacing x with some non-linear function of the input $\phi(x)$. This is known as basis function expansion. For example, if x is in $\mathbb{R}$, we can take $\phi(x) = (1, x, \dots, x^{k-1})$ (x is of dimension 1 and $\phi(x)$ is of dimension k).

Logistic regression. We want to generalize linear regression to a case where we do a binary classification.

(1) We suppose the law of y knowing x and w is Bernoulli with parameter $\mu(x)$ (denoted by $\mathscr{B}(\mu(x))$. The parameter $\mu(x)$ is defined below.
(2) We compute a linear combination of the input and apply a function that ensures that $\mu(x)$ is in $[0,1]$. That is we define $\mu(x) = \text{sigm}(w^T x)$, with a well-chosen function sigm. It is customary to use the sigmoid function for sigm:

$$\text{sigm}(\eta) := \frac{1}{1+e^{-\eta}} = \frac{e^{\eta}}{1+e^{\eta}}$$

(see Figure 1.4.4).

We get that the law of y knowing x and w is Bernoulli with parameter $\text{sigm}(w^T x)$. This is called logistic regression (although this is a case of classification, not regression). Observe that if we know w, we can then make a prediction: $\mathbb{P}(y = 1) = \text{sigm}(w^T x)$. So it remains to us to learn w on a training set.

EXAMPLE 1.7. Here we take x and y to be of dimension 1. For i in $\{1, 2, \dots, N\}$, we have

$$\begin{cases} x_i = \text{SAT score of student } i, \\ y_i = 0 \text{ or } 1 \text{ (fail or pass 1st year of college).} \end{cases}$$

On Figure 1.4.5, the pink dots show the training data. From this data, we get an estimate $\widehat{w}$ of w (see the supplementary material for the details of the computation).

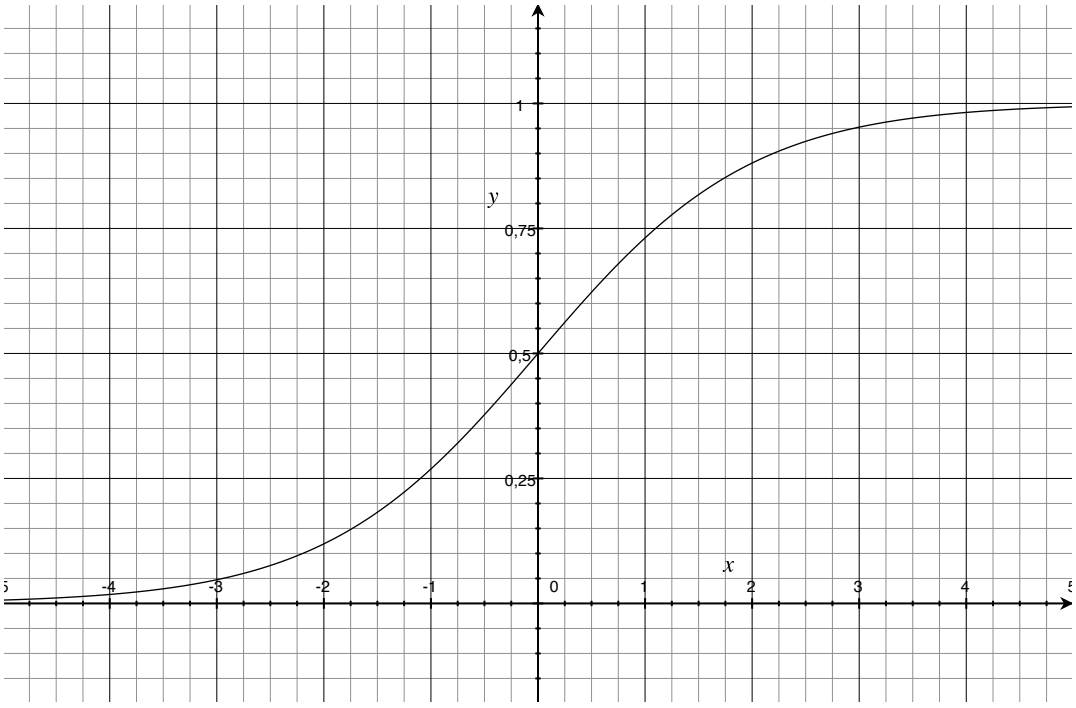


FIGURE 1.4.4. Sigmoid function

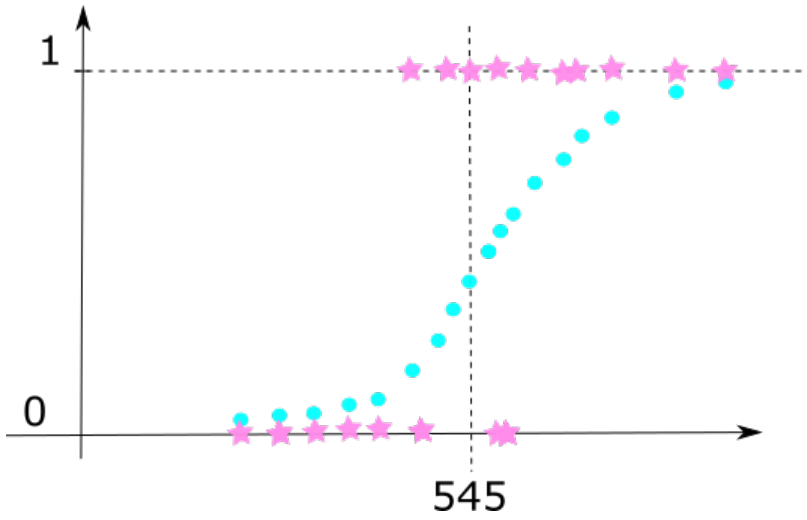


FIGURE 1.4.5. Logistic regression

We simply set

$$\widehat{w} = \arg\max_{w} \sum_{i=1}^{N} (y_i - \text{sigm}(w^T x_i))^2 ,$$

and then find $\widehat{w}$ numerically, using R.

EXAMPLE 1.8. Then, when a new x comes by, we follow the decision rule:

$$\widehat{y}(x) = 1 \Longleftrightarrow \text{sigm}(w^T x) > 0.5 .$$

On the graph, we have plotted $x \mapsto \text{sigm}(w^t x)$ in blue. We see that $\text{sigm}(w^T x) = 0.5$ for $x = 545$. We set $x^* = 545$. We draw a vertical line at $x = x^*$ and we call it the decision boundary. Everything on the left

of this line is classified as 0 and everything on the right of this line is classified as 1. The decision-rule has a non-zero error rate even on the training set. This is because the data is not linearly separable, i.e. there is not vertical straight line separating the zeros and the ones. Later, we will see examples with a non-linear decision boundary.

DEFINITION 1.9. If we have a finite set A and a finite set B in $\mathbb{R}^d$. We say that A and B are linearly separable if there exists an hyperplane H (of dimension $d-1$) separating A and B. If we parameter H by

$$H = \{x \in \mathbb{R}^d : \langle a, x\rangle + b = 0\},$$

(where $a \in \mathbb{R}^d$, $b \in \mathbb{R}$, $\langle \ldots, \ldots \rangle$ is the scalar product) then, for all $x \in A$, $\langle x, a\rangle + b$ has always the same sign and for all $x \in B$, $\langle x, a\rangle + b$ has always the other sign.

In the above example, $d = 1$ so an hyperplane separating two sets of points is made of ... a single point. In Figure 1.4.5, we have a mixed view with the input on the x-axis and the output (0 or 1 on the y-axis).

1.4.4. Over-fitting and under-fitting. We need to be careful that we do not overfit the data. If we model every minor variation of the input, which is more likely to be noise than signal, it will result in poor predictions.

EXAMPLE 1.10. We look at Figure 1.4.6, where we have two type of points (blue and red). What we

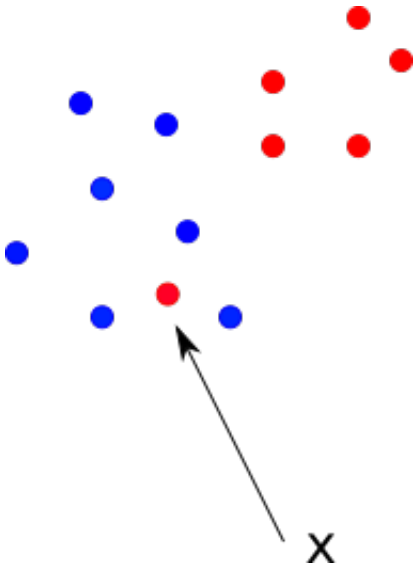


FIGURE 1.4.6. Over-fitting example

see here is our training set $\mathscr{D}$. Suppose we train a KNN classifier with $K = 1$. This has the advantage that each point in the training set is classified correctly. The point x is its nearest neighbor, so the KNN will classify it as red. Also, the KNN will classify any point in the vicinity of x as red. But the presence of the red point x amongst blue point is likely due to randomness, meaning that we would rather classify the area around x as blue. This is a classical example where the model fits perfectly the data, which will result in poor predictions.

On the other hand, if our model does not fit at all the data, we are under-fitting. It also results in poor predictions.

EXAMPLE 1.11. Now we look at Figure 1.4.7. Here, we have points painted in blue and points painted in red, distributed in $\mathbb{R}^2$. For a KNN classifier, we can color in red the area where the classifier predicts a probability to be red bigger than one half, and we can color in blue the area where the classifier predicts a probability to be blue bigger than one half. We did this for various K (1, 3, 9).

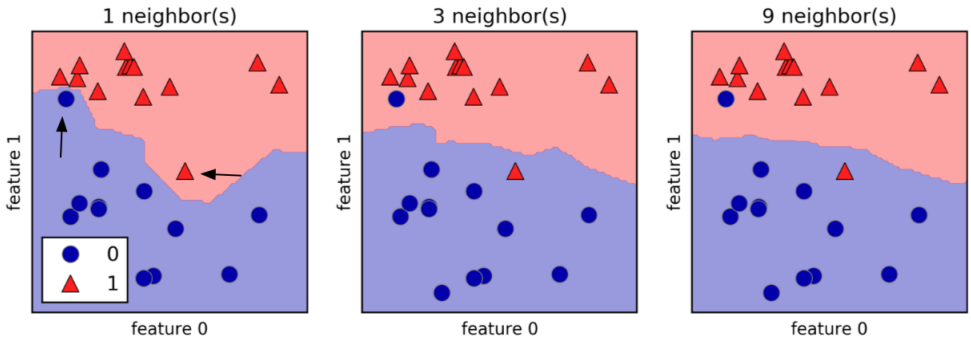


FIGURE 1.4.7. Another over-fitting example ([**MMG18**]).

For $K = 1$, the model fits perfectly the data as every red point is in the red area and every blue point is in the blue area. Let us call x_0 the blue point indicated with an arrow in the drawing on the left. As this guy is near a bunch of red points, it is likely that it is an anomaly i n the data and we think that any point in the vicinity is most likely blue. So the KNN with $K = 1$ performs poorly in the neighborhood of this point.

We could perform a similar analysis with the red point indicated with an arrow. in the drawing on the left.

We see that the KNN for $K = 3$ or $K = 9$ behaves better.

EXAMPLE 1.12. Suppose we have trained our classifier. We test it on the test set, where it has a 91% accuracy. We test it on the training set, where it has a 99% accuracy. This is a clear sign of over-fitting (it fits very well on the data we used to train it and fits significatively less well on the test set).

1.4.5. Model selection. The question here is: how do we choose our model? Suppose we have a data set $\mathscr{D} = \{(x_1, y_1), \ldots, (x_N, y_N)\}$. We first compute the misclassification rate on the training set for each proposed method. This error is, for the classifier f,

$$E(f) = \frac{1}{N} \sum_{i=1}^{N} \mathbb{1}_{f(x_i) \neq y_i}. \tag{1.1}$$

That is we compute the number of times the classifier f makes a mistake when applied to the points if $\mathscr{D}$.

But we can do better. What we really want is the generalization error, which is the expected value of the misclassification rate when averaged over future data. We do not know which data we will have to classify so we do something mimicking the incoming of new data: we split our data into a training set and a test set (or validation set). We train our model on the training data and then we compute the prediction error on the test set (just like in Equation (1.1), but we sum only over the test set).

In the case of the KNN classifier, we get a different classifier for each K. Hopefully, the prediction error will look like the graph of Figure 1.4.8: a U-shaped curve. We then pick K such that it minimizes the prediction error on the test set. When K is picked, we refit the model to all the data.

The rule of thumb is the following: 80% of the data for the training set and 20% for the validation set. But we run into problems if the data is small, because the model will not have enough data to train on, or will not have enough data to make a reliable estimate of the future performance.

The solution is to use cross-validation: split the data into P folds. Then, for each fold $p \in \{1, 2, \ldots, P\}$, do the training on all the folds except the p-th. Then compute the error averaged on all the folds. It is common to use $P = 5$. We get a method called leave one out cross-validation (LOOCV).

Choosing K for a KNN classifier is a special case of a more general problem known as model selection, where we have to choose between models with different degrees of flexibility.

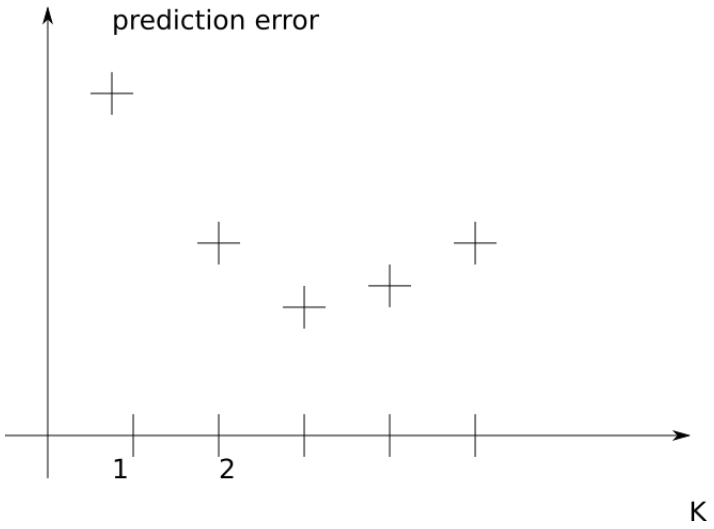


FIGURE 1.4.8. U-shaped curve

1.5. Examples in R

1.5.1. Iris dataset. We are interested in the iris dataset. For each flower, we have four measurements and the name of the specie. We begin our code with (observe that the # is for comments)

```
data(iris)
str(iris)
```

so that we load the data and observe its structure. The computer says that in the specie compartment, there are three "levels", meaning there are three possible species. We then load various packages

```
install.packages("e1071")
install.packages("caTools")
install.packages("class")
library(e1071)
library(caTools)
library(class)
data(iris)
head(iris)
```

(observe that the command `head` gives us the beginning of the data set). For each flower, we have four measurements and the specie (see Figure 1.5.1to understand the measurements). We then split the data into a training set and a test set

```
split <- sample.split(iris, SplitRatio = 0.7)
train_cl <- subset(iris, split == "TRUE")
test_cl <- subset(iris, split == "FALSE")
```

Observe that the `split` command attaches labels TRUE and FALSE, in each subset (TRUE and FALSE) the ratios of species are preserved . This fact is later used to actually split the data. We then re-scale the data (see Remark 1.6)

```
train_scale <- scale(train_cl[, 1:4])
test_scale <- scale(test_cl[, 1:4])
```

We try a first classification with $K = 1$.

```
classifier_knn <- knn(train = train_scale, test = test_scale,
                      cl = train_cl$Species, k = 1)
```

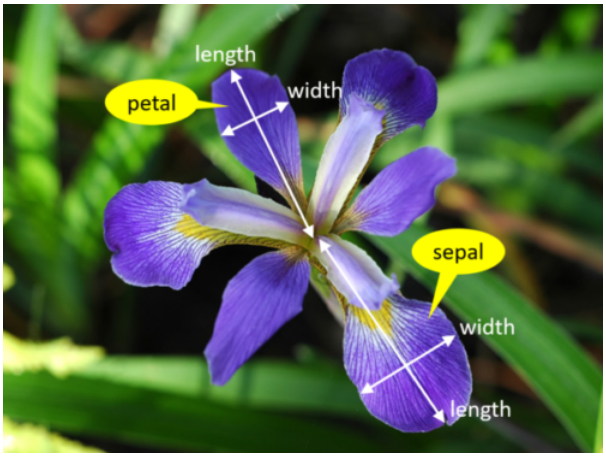


FIGURE 1.5.1. Measurements.

```
classifier_knn
cm <- table(test_cl$Species, classifier_knn)
cm
```

Here `cm` is the confusion matrix: the line indexes are the actual classes, the column index are the predicted classes and in the matrix, at the coordinates (i, j), we find the number of cases for actual class i and predicted class j.

We then compute, for various K, the accuracy of the KNN classifier.

```
lk=c(1,3,5,7,15,19,30,40)
er=c(1:6)
# lk and er are line vectors
A=rbind(lk,er)
# rbind glues two matrices
# the matrix A will contain various K and the related accuracy
for (i in 1:6)
{
  classifier_knn<-knn(train=train_scale, test = test_scale,
                      cl = train_c$Species,k = A[1,i])
  misClassError<-mean(classifier_knn!=test_cl$Species)
# Here we compute the number of times the prediction differs from
   the actual class
#(and normalize it to get a mean).
  print(paste('Mean error =', misClassError))
  A[2,i]<-misClassError
}
```

We can then plot the accuracy vs. K.

```
plot(lk,A[2,],type='b',col='red',xlab='K',ylab='Mean error')
```

From which we get Figure 1.5.2. The curve is not (almost) a nice convex curve. We see that K below 20 provides the smallest error.

When wanting to make new predictions, we will not forget to re-fit the model on all the data (with $K = 10$).

Unfortunately

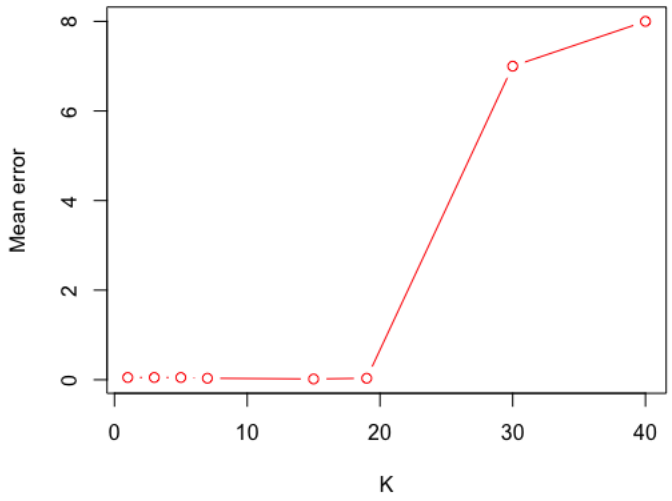


FIGURE 1.5.2. Mean error vs. K

1.5.2. Pima Indian diabetes dataset. Now, you can try yourself at the diabetes dataset. In this dataset, you have, for each individuals: eight variables and one outcome (1 if the individual has diabetes, 0 if not). The file is named `diabetes.csv`. To load it into R, you need the `read.csv()` command.

Which is the best K?

1.5.3. Summary of useful R commands. Split randomly a set called `data`.

```
library(catools)
split<-sample.split(data, SplitRatio=0.8)
data1<-subset(data, split="TRUE")
data2<-subset(data, split="TRUE")
```

Re-scale data (here we choose the columns we want to re-scale).

```
data_scale=scale(data[,2:6])
```

The KNN classifier (training data=`train_data`, test data=`test_data`, labels in `$label`)

```
library(e1071)
classif<-knn(train=train_data, test=test_data, cl=train_data$label, k=3)
```

This is a powerful command which gives you, among other things, the predicted labels for the test data.

Get the confusion matrix for `classif`:

```
table(test_data$label, classif)
```

and compute the missclassification error

```
missClassError<-mean(classif!=test_data$label)
```

(we compute the mean of the number of times our prediction is correct).

CHAPTER 2

Linear regression

2.1. Introduction

2.1.1. Generalities. We are going to study multiple linear regression. We know the concept of simple linear regression where a single input variable X is used to model the output variable Y (seen in Section 1.4.3). In many applications, there is more than one factor that influences the response. Multiple regression models thus describe how a single output variable Y depends linearly on a number of input variables. Here are some examples.

- The selling price of a house can depend on the desirability of the location, the number of bedrooms, the number of bathrooms, the year the house was built, the square footage of the lot and a number of other factors.
- The height of a child can depend on the height of the mother, the height of the father, nutrition, and environmental factors.
- Data scientists in the NBA might analyze how different amounts of weekly yoga sessions and weightlifting sessions affect the number of points a player scores. They might fit a multiple linear regression model using yoga sessions and weightlifting sessions as the predictor variables and total points scored as the response variable.

2.1.2. Housing values. Let us have a look at the Boston Housing dataset in Figure 2.1.1 (Housing Values in Suburbs of Boston) It contains various numbers (each line is relative to an area) such as

crim	zn	indus	chas	nox	rm	age	dis	rad	tax	ptratio	b	lstat	medv
0.00632	18	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
0.02731	0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
0.02729	0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
0.03237	0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
0.06905	0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2
0.02985	0	2.18	0	0.458	6.430	58.7	6.0622	3	222	18.7	394.12	5.21	28.7

.....

FIGURE 2.1.1. Boston Housing Dataset

- `rm` average number of rooms per dwelling.
- `age` proportion of owner-occupied units built prior to 1940.
- `dis` weighted mean of distances to five Boston employment centers.
- ...
- `medv` median value of owner-occupied homes in $1000s (this is the target variable, i.e. the one we would like to predict)

Imagine now you want to build a new suburb in a formerly industrial area. You would be interested in predicting the selling prices of the houses in this new suburb, for your own convenience (because you want to know how much money can be made there) and as a selling argument ("buy it now at this price, re-sell it at a higher price").

2.1.3. Relationship between the input data and the selling price (continuation of the example above). We want to model the selling price as a function of the various input parameters. Any real-estate agent will tell you that if the variable `crim` (per capita crime rate by town) is high, it will lower the prices, and if the variable `rm` (average number of rooms per dwelling) is high, it will raise the prices. If

we call $(x_1,\dots,x_n)$ the input variables, with, say, x_1 =`crim`, x_2 =`rm`, ... then y =`medv` should be a mixture of x_1, x_2, ... So we imagine

$$y = w_1x_1 + w_2x_2 + \cdots + w_nx_n \tag{2.1}$$

with $w_1 < 0$ and $w_2 > 0$ in order to be consistent with our above analysis.

REMARK 2.1. One might argue that, for example, the variable `crim` lower the prices *below* the average value. So an ideal model would be: $y = w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n$ with, say, w_0 the average value of housing around all the Boston suburbs. In this case, we add an input variable x_0, which is always equal to 1 and write $y = w_0x_0 + w_1x_1 + \cdots + w_nx_n$. Thus our equation has the same form as Equation (2.1) above.

There is no reason that one could find a model that predicts perfectly the housing prices. So we decide to add a part that cannot be controlled:

$$y = w_1x_1 + \cdots + w_nx_n + \epsilon \tag{2.2}$$

Here ϵ is a random noise, say a $\mathcal{N}(0,\sigma^2)$ variable. This is the part of the final price that is due to chance. By chance, we mean all the factors that we cannot predict/did not include in our model such as the mood of the buyers or the presence of a nice rosebushes in a street, and so on.

2.1.4. Fitting the model. Once we have made up our mind about the model, we have to fit the parameters $w_1,\dots,w_n,\sigma$ on our data. If these parameters are good they should perform well on the data we have. If we call $x_1^{(k)},\dots,x_n^{(k)}$ the data we have in the k-th line of Figure 2.1.1, then our prediction is $w_1x_1^{(k)} + \cdots + w_nx_n^{(k)}$ and it should not be "too far" from the output variable $y^{(k)}$. The difference between these two quantities $(y^{(k)} - w_1x_1^{(k)} + \cdots + w_nx_n^{(k)})$ is called the prediction error. We want all the prediction errors to be small but this might not be possible. So we settle for a criterion like: "the average error should be small" or "the sum of the absolute values of the prediction errors should be as small as possible". In the next sections, we will see mathematical formulation of the problem.

2.2. Model specification

The input is a vector:

$$x = (x_1, x_2, \dots, x_D)$$

and the output is

$$\begin{aligned} y &= \sum_{i=1}^{D} w_i x_i + \epsilon \\ &\quad \text{(we use a matrix formulation)} \\ &= w^T x + \epsilon, \end{aligned} \tag{2.1}$$

where ϵ is a Gaussian noise with mean 0 and variance σ^2. The model has a parameter $\theta = (w,\sigma^2)$ (here, we want to have one parameter so we put all the scalar parameters into one big vector). By scalar, we mean "of dimension one".

2.3. Maximum likelihood estimator

A common way to estimate the parameter θ is the maximum likelihood estimator (MLE) . We suppose we have various inputs and outputs: $\mathcal{D} = (x^{(i)}, y^{(i)})_{1\le i\le N}$ ($\mathcal{D}$ is for data). Each time I have an input, the output is generated with a noise independent of the other noises (and always Gaussian with mean 0 and variance σ^2. For each i, the density of $y^{(i)}$ knowing $x^{(i)}$ and θ is

$$p(y^{(i)}|x^{(i)},\theta) = \frac{\exp\left(-\frac{(w^Tx^{(i)}-y^{(i)})^2}{2\sigma^2}\right)}{\sqrt{2\pi\sigma^2}}.$$

So we can write the density of $y^{(1,\dots,N)} = (y^{(1)},\dots,y^{(N)})$ knowing $x^{(1,\dots,N)} = (x^{(1)},\dots,x^{(N)})$ and θ as a product :

$$p(y^{(1,\dots,N)}|x^{(1,\dots,N)},\theta)=\prod_{i=1}^{N}\frac{\exp\left(-\frac{(w^Tx^{(i)}-y^{(i)})^2}{2\sigma^2}\right)}{\sqrt{2\pi\sigma^2}}.$$

The MLE is the parameter θ maximizing $p(y^{(1,\dots,N)}|x^{(1,\dots,N)},\theta)$, which we can write:

$$\widehat{\theta}=\arg\max_{\theta} p(y^{(1,\dots,N)}|x^{(1,\dots,N)},\theta).$$

The most convenient is to look for the minimum of the negative log-likelihood (NLL):

$$\begin{aligned}\text{NLL} &= -\log\prod_{i=1}^{N}p(y^{(i)}|x^{(i)},\theta)\\ &= \frac{1}{2\sigma^2}\underbrace{\sum_{i=1}^{N}(w^Tx^{(i)}-y^{(i)})^2}_{=:\text{RSS}(w)}+\frac{N}{2}\log(2\pi\sigma^2),\end{aligned}$$

where RSS is the residual sum of squares.

Suppose σ^2 is known, then we look for the vector w minimizing the RSS. This method is known as the least squares regression.

EXAMPLE 2.2. Let $D=1$. In this case, we have inputs $x^{(1)},\dots,x^{(N)}$ in $\mathbb{R}$ and outputs $y^{(1)},\dots,y^{(N)}$ in $\mathbb{R}$. We suppose that σ^2 is known. We rewrite the data as $\overline{x}^{(1)}=(1,x^{(1)}),\overline{x}^{(2)}=(1,x^{(2)}),\dots$. We suppose that for all i, $y^{(i)}=w^T\overline{x}^{(i)}+\epsilon_i$ (with independent noises ϵ_i) where $w=\begin{pmatrix}w_0\\ w_1\end{pmatrix}$ (observe that this model is non-linear in the data $(x^{(i)})$, we made it linear in the data $\overline{x}^{(i)}$ just by adding a component). So we are looking for parameters w_0, w_1 such that the green line is nearest to the points $(x^{(i)},y^{(i)})$ (see Figure 2.3.1, w_0 is the intercept and w_1 is the slope). More precisely, we want the sum of the squares of the

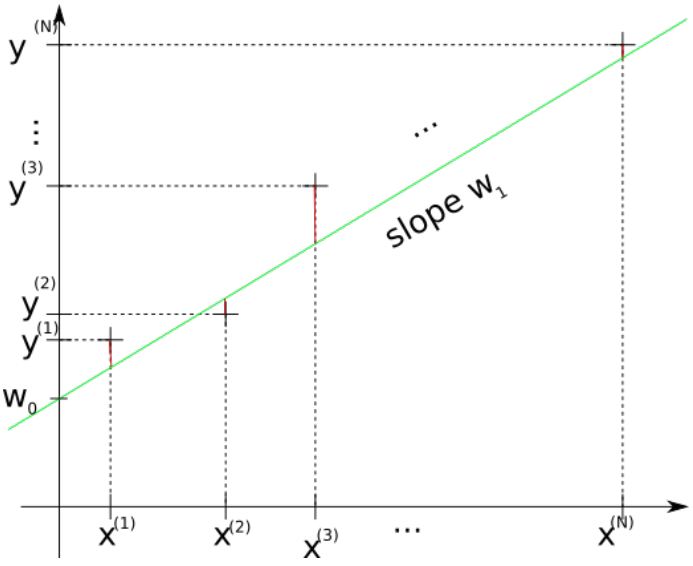


FIGURE 2.3.1. Linear regression

length in red to be as small as possible.

In practice, when we want to find $\widehat{\theta}$, we have to study a function of $D+1$ parameters.

2.3.1. Derivation of the MLE (σ^2 known). We set

$$y = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(N)} \end{pmatrix} \text{(column vector)},$$

$$w = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_D \end{pmatrix} \text{(column vector)},$$

$$X = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_D^{(1)} \\ \vdots & & & \vdots \\ x_1^{(N)} & x_2^{(N)} & \dots & x_D^{(N)} \end{bmatrix} = \begin{bmatrix} \frac{(x^{(1)})^T}{\vdots} \\ \frac{\vdots}{(x^{(N)})^T} \end{bmatrix},$$

where $(x^{(1)})^T, \dots, (x^{(N)})^T$ are line vectors (T is the transposition, and $x^{(1)}, \dots$ are column vectors). Then we can write (forgetting the σ^2)

$$\begin{aligned} \text{NLL}(w) &= \frac{1}{2}(y - Xw)^T(y - Xw) \\ &= \frac{1}{2}w^T(X^TX)w - w^T(X^Ty) + \frac{1}{2}y^Ty, \end{aligned} \tag{2.1}$$

where

$$\begin{aligned} X^TX &= \begin{bmatrix} x^{(1)} & | & \dots & | & x^{(N)} \end{bmatrix} \times \begin{bmatrix} \frac{(x^{(1)})^T}{\vdots} \\ \frac{\vdots}{(x^{(N)})^T} \end{bmatrix} \\ &= \sum_{i=1}^{D} x_i x_i^T, \end{aligned}$$

$$X^Ty = \sum_{i=1}^{N} x^{(i)} y^{(i)}.$$

To compute the gradient, we need to be able to compute the gradient of: $w \mapsto w^TAw$ (with A a symmetric matrix). We write $A = ((a_{i,j}))_{1 \le i,j \le N}$. Then we have

$$w^TAw = \sum_{i=1}^{D}\sum_{j=1}^{D} w_i a_{i,j} w_j,$$

whose gradient[1] is a vector with the following at line k:

$$\begin{aligned} \frac{\partial}{\partial w_k}\left(\sum_{i=1}^{D}\sum_{j=1}^{D} w_i a_{i,j} w_j\right) &= \sum_{j=1}^{D} a_{k,j} w_j + \sum_{i=1}^{D} w_i a_{i,k} \\ &= 2\sum_{j=1}^{D} a_{k,j} w_j. \end{aligned}$$

So the wanted gradient is: $2Aw$.

The gradient of $w \mapsto w^Tz$ (constant vector z) is a vector with the following at line k:

$$\frac{\partial}{\partial w_k}\left(\sum_{i=1}^{D} w_i z_i\right) = z_k.$$

In this case the gradient is the constant vector z.

In conclusion, we get the gradient (∇ means "gradient")

$$\nabla \text{NLL}(w) = (X^TX)w - X^Ty.$$

[1] For $f : \mathbb{R}^D \to \mathbb{R}$, $x = (x_1, \dots, x_D) \mapsto f(x)$, the gradient of f in x is $(\frac{\partial f}{\partial x_1}(x), \dots, \frac{\partial f}{\partial x_D}(x))^T$.

Under the assumption that X^TX is invertible (see Exercise 2.1 below), the critical point[2] is

$$\widehat{w} = (X^TX)^{-1}X^Ty. \tag{2.2}$$

2.3.2. Geometric interpretation. We assume $N > D$. We set[3]

$$E = \text{Span}\left(\left(\begin{array}{c} x_i^{(1)} \\ \vdots \\ x_i^{(N)} \end{array}\right), 1 \leq i \leq D\right).$$

The subspace E is of dimension $\leq D$. Let us suppose that its dimension is exactly D. We set (for $1 \leq i \leq D$)

$$\widetilde{x}_i = \left(\begin{array}{c} x_i^{(1)} \\ \vdots \\ x_i^{(N)} \end{array}\right).$$

What we do in the linear regression is that we seek a vector $\widehat{y}$ in E which is as close as possible to y (this can be understood by looking at Equation (2.1)). If we write $z = w_1\widetilde{x}_1 + \cdots + w_D\widetilde{x}_D$ (a generic element of E), we want to minimize the norm of $y - z$. This is equivalent to wanting $y - z$ orthogonal to E.

So, if we write $\widehat{y}$ for the best z possible ($z = w_1\widetilde{x}_1 + \cdots = Xw$), we get: $\widetilde{x}_j^T(y - \widehat{y}) = 0$ ($1 \leq j \leq D$). Hence: $\widetilde{x}_j^T(y - Xw) = 0$ ($1 \leq j \leq D$). So $X^T(y - Xw) = 0$. And so $w = (X^TX)^{-1}X^Ty$, $\widehat{y} = Xw = X(X^TX)^{-1}X^Ty$. This is the orthogonal projection of y onto E. The projection matrix $P := X(X^TX)^{-1}X^T$ is called the hat matrix because it "puts a hat on y".

EXERCISE 2.1. Suppose $D \leq N$ and E is of dimension D exactly, show that X^TX is invertible.
Answer: We have

$$X = \left[\begin{array}{ccccc} \widetilde{x}_1 & | & \ldots & | & \widetilde{x}_D \end{array}\right].$$

So any element of E is of the form Xw ($w \in \mathbb{R}^D$). For any y in $\mathbb{R}^N$, $\exists! w$ such that $y - Xw$ is orthogonal to E which we can write: $\exists! w$ such that $X^T(y - Xw) = 0$. This last equality can also be written: $X^TXw = X^Ty$. We have supposed that $\text{Rank}(X) = D$, so $\text{Rank}(X^T) = D$, which means that $\text{Span}(x^{(1)}, \ldots, x^{(N)}) = \mathbb{R}^D$. So any element of $\mathbb{R}^D$ can be written X^Ty. So we have proved, that for any element X^Ty of $\mathbb{R}^D$, there exists w such that $X^TXw = X^Ty$. Which is exactly saying that X^TX is invertible.

EXERCISE 2.2. Data scientists for professional sports teams often use linear regression to measure the effect that different training regimens have on player performance. For example, data scientists in the NBA might analyze how different amounts of weekly yoga sessions and weightlifting sessions affect the number of points a player scores. They might fit a multiple linear regression model using yoga sessions and weightlifting sessions as the predictor variables and total points scored as the response variable. The regression model would take the following form:

$$\text{points scored} = \beta_0 + \beta_1 \times (\text{yoga sessions}) + \beta_2 \times (\text{weightlifting sessions}).$$

The coefficient β_0 would represent the expected points scored for a player who participates in zero yoga sessions and zero weightlifting sessions. The coefficient β_1 would represent the average change in points scored when weekly yoga sessions is increased by one, assuming the number of weekly weightlifting sessions remains unchanged. The coefficient β_2 would represent the average change in points scored when weekly weightlifting sessions is increased by one, assuming the number of weekly yoga sessions remains unchanged. Depending on the values of β_1 and β_2, the data scientists may recommend that a player participates in more or less weekly yoga and weightlifting sessions in order to maximize their points scored.

We can see the data we have in Table 2.3.1. We perform a multiple regression on this data. First, we load the data into a data-frame:

```
basket<-data.frame(yoga=c(10,25,5,30),muscu=c(40,25,35,10),points=c(6,6,2,5))
```

Then we perform the regression itself:

[2] A critical point is a point where the gradient is zero. Here, we have only one critical point.
[3] Span ="vector space spanned by ..."

Player	Yoga sessions during last month (hours)	Weightlifting sessions (hours)	Number of points gained during the match (after the month of preparation)
1	10	40	6
2	25	25	6
3	5	35	2
4	30	10	5

TABLE 2.3.1. Historic data.

```
regmulti<–lm(points~yoga+muscu, data=basket)
```

We can see the result by using the command

```
summary(regmulti)
```

Questions:

(1) Is it more profitable to add yoga or weight-lifting in the time-table of the players ?
(2) A new player is big on weight-lifting, he totalizes 50 hours of weight-lifting and 0 hours of yoga. How many points can he be expected to score during the next match ?
Use the commands:

```
new=data.frame(yoga=0,muscu=50)
predict.lm(regmulti,new)
```

2.4. Ridge regression

The MLE can overfit because it picks the parameters that are good for the training data. But the training data can be noisy so there is no reason to pick the best parameters for the training data.

The alternative to linear regression is to use a Bayesian statistics approach. We suppose

$$y^{(i)} = w_0 + w^T x^{(i)} + \epsilon_i$$

for all i in $\{1,2,\dots,N\}$ (caution: we have added a parameter w_0 when compared to Equation (2.1), this has no big impact). We assume an a priori distribution on the $w_0, w_1, \dots, w_D$ (corresponding to prior knowledge/preconceived idea/...). Here, we have in mind that they should not be too big so we choose the following a priori:

$$p(w) = \prod_{j=0}^{D} \mathcal{N}(w_j|0;\tau^2),$$

meaning[4] each w_j is independent of the others and of Gaussian law with mean 0 and variance τ^2.

The likelihood is the density of $y_1,\dots,y_D$ under the assumption that we know $w_0,\dots,w_D$:

$$p((y^{(1)},\dots,y^{(N)})|(x^{(1)},\dots,x^{(N)}),\theta=(w_0,\dots,w_D)) = \prod_{i=1}^{N} \mathcal{N}(y^{(i)}|w_0 + w^T x^{(i)},\sigma^2),$$

which means we have a product of Gaussian densities with mean $w_0 + w^T x^{(i)}$ ($1 \le i \le N$) and variance σ^2.

The a posteriori distribution is the product of the a priori distribution and the likelihood, renormalized to be a probability distribution (in $w = (w_0, w_1, \dots, w_D)$) We want to find the Maximum a Posteriori (MAP) (the maximum of the a posteriori distribution. So we look for w maximizing the log of the a posteriori distribution (we take the log and throw away the normalization):

$$\sum_{i=1}^{N} \log(\mathcal{N}(y^{(i)}|w_0 + w^T x^{(i)},\sigma^2)) + \sum_{j=0}^{D} \log(\mathcal{N}(w_j|0,\tau^2)).$$

[4]Caution: $\mathcal{N}$ usually means the Gaussian law, here it means the Gaussian density.

This is the same as minimizing

$$\sum_{i=1}^{N} \frac{(y^{(i)} - w_0 - w^T x^{(i)})^2}{2\sigma^2} + \sum_{j=0}^{D} \frac{w_j^2}{2\tau^2},$$

or (equivalently)

$$\sum_{i=1}^{N} (y^{(i)} - w_0 - w^T x^{(i)})^2 + \lambda(w_0^2 + w^T w),$$

with $\lambda = \sigma^2/\tau^2$.

💡 We can interpret the first term as the NLL we have seen before and the second term as a complexity penalty (we do not want the $|w_j|$ to be too big). This idea that we can force the solution to have small absolute values $|w_j|$ by adding a penalty term is very powerful and we will it at many places in this course. This technique is known as ridge regression or penalized least squares. In general, adding a Gaussian prior to the parameters of a model to encourage them to be small is called l_2-regularization of weight-decay.

When it comes to interpreting which variables are important in our model, the ridge regression has the disadvantage that it produces small coefficients w_k which are not zero. The next algorithm (LASSO) overcomes this disadvantage.

2.5. The LASSO (Least Absolute Shrinkage and Selection Operator)

Again, we suppose

$$y^{(i)} = w_0 + w^T x^{(i)} + \epsilon_i$$

for all i in $\{1, 2, \ldots, N\}$. We define a new estimator:

$$\widehat{w}_{\text{LASSO}} = \arg\min_{w} \{ \sum_{i=1}^{N} (y^{(i)} - w_0 - w^T x^{(i)})^2 : w_0, w \text{ such that } \sum_{j=0}^{D} |w_j| \leq t \},$$

where we have to choose t.

💡 Making t small will cause some of the coefficients w_0, w_1, ... to be exactly zero. This might be a way to set apart two classes of variables: those who are not important for the predictions (with an index i such that w_i is zero) and the others.

If t is chosen bigger than $t_0 = \sum_{j=0}^{D} |\widehat{w}_j^{\text{LS}}|$ (where the $\widehat{w}_j^{\text{LS}}$ are the coefficients found with the least square regression) then $\widehat{w}_{\text{LASSO}} = \widehat{w}^{\text{LS}}$.

The parameter t should be adaptively chosen to minimize an estimate of expected prediction error (again, we have to split our data into a training set and a test set).

The LASSO and ridge are part of what is called shrinkage methods.

2.6. Advantages/disadvantages of linear regression

Advantages.

- Easy to implement, easy to train, easy to interpret.
- Linear regression is prone to over-fitting but this can be overcome by using ridge regression or the LASSO.

Disadvantages.

- The main problem of linear regression is the assumption that the output variable depends linearly on the input variables.
- It is very sensitive to noise. If the number of observations is less than the number of features (i.e. the dimension of the input variables), then the regression should not be used (it will overfit, taking the noise into account where it should ignore it).
- Prone to outliers (anomalies). These should be analyzed and removed before proceeding with the regression.

2.7. Examples in R

2.7.1. BigMart data set. In the BigMart data set[5], we have informations about 1559 products sold in 10 different stores. Some features are identified (identifier, weight, ...). We would like to predict the outlet sales (sales of the product in the particular store).

We first download the packages we will need.

```
library(data.table) # used for reading and manipulation of data
library(dplyr)      # used for data manipulation and joining
library(glmnet)     # used for regression
library(caret)      # used for modeling
library(elasticnet) # used for regression
```

Remember that you have to install the packages beforehand (using `install.packages(''glmnet'')`), but you have to do it only once (whereas you should do the `library(...)` at the beginning of each program).

We load the data sets.

```
train=read.csv("Train.csv")
test=read.csv("Test.csv")
```

In the test set, there is no "outlet sales" column so we add one. And then we glue the two sets together.

```
test$Item_Outlet_Sales=NA
combi = rbind(train, test)
```

Now, some values are missing (weights). We replace the NA value by a mean over the items with the same identifier (meaning this is the same product), excluding those for which the value is also NA (this is the meaning of the `na.rm = TRUE`).

```
missing_index = which(is.na(combi$Item_Weight))
for(i in missing_index)
{
  item = combi$Item_Identifier[i]
  combi$Item_Weight[i]=
                mean(combi$Item_Weight[combi$Item_Identifier==item]
                                                ,na.rm = TRUE)
}
```

Some values are categorical and easy to convert into numbers ({"small","medium", ...} is converted into 0,1,...). We use here the `ifelse` command which works as follows:
`ifelse(x=''small'',0,1)` returns 0 if `x=''small''` and otherwise returns 1 (this is an example). Remember that `combi` is a table. By doing `combi$Outlet_Size=...`, we modify the whole "Oulet_Size" column.

```
combi$Outlet_Size_num <- ifelse(combi$Outlet_Size=="Small",0,
                          ifelse(combi$Outlet_Size=="Medium",1, 2))
combi$Outlet_Location_Type_num<-
        ifelse(combi$Outlet_Location_Type=="Tier3",0,
      ifelse(combi$Outlet_Location_Type=="Tier2",1,2))
```

We erase columns which we think are not relevant to predict sales (this might be subject to discussion).

```
combi$Outlet_Size<-NULL
combi$Outlet_Location_Type<-NULL
combi_temp=combi
```

[5] `https://www.kaggle.com/brijbhushannanda1979/bigmart-sales-data`

```
combi_temp$Item_Identifier=NULL
combi_temp$Outlet_Establishment_Year=NULL
combi_temp$Item_Type=NULL
```

Now, we have to deal with variables such as item type. We could transform them into numerical values by picking different numbers for each of them, "dairy" becomes 1, "meat" becomes 2, and so on. But this is too artificial (and will lead to poor predictions). We prefer to create dummy variables (see Section 2.8 for a detailed explanation on dummy variables).

```
ohe_1 = dummyVars("~.", data = combi_temp, fullRank = T)
ohe_df = data.table(predict(ohe_1, combi_temp))
combi = cbind(combi[, "Item_Identifier"], ohe_df)
```

We are now going to scale and center the data. This has no impact on our prediction but this is important to have all the data of the same order of magnitude when we compare the coefficients of the vector w.

```
num_vars = which(sapply(combi, is.numeric))
# index of numeric features
num_vars_names = names(num_vars)
combi_numeric=combi[,setdiff(num_vars_names,"Item_Outlet_Sales"),
                    with = F]
prep_num=preProcess(combi_numeric,method=c("center","scale"))
combi_numeric_norm = predict(prep_num, combi_numeric)
combi[,setdiff(num_vars_names,"Item_Outlet_Sales"):=NULL]
combi = cbind(combi, combi_numeric_norm)
```

We use here the command `sapply` (applies a function to each part a `combi`, the result is a boolean). The command `setdiff(A,B)` returns the set of elements of A which are not in B. According to the CRAN manual[6], the `with=F` option is necessary when we want to exclude columns. Observe the syntax of the "preprocessing": we use the command `preProcess` to center and scale, and then use `predict` to actually get our processed data.

We split again the data between a training set and a test set.

```
train = combi[1:nrow(train)]
test = combi[(nrow(train) + 1):nrow(combi)]
test[, Item_Outlet_Sales := NULL]
```

We split the training data into a set with only numerical data (without the sales) and a set with the sales.

```
xt=train
xt$V1=NULL
xt$Item_Outlet_Sales=NULL
yt= train$Item_Outlet_Sales
```

Now, we can train our LASSO model and make predictions. We choose a grid `lambda_grid` in which we will look for the best parameter λ (defined previously). The variable `control` is such that we use a cross-validation method with five folds. Observe that the parameter λ has to be named `fraction` in the code.

```
control = trainControl(method ="cv", number = 5)
lambda_grid=c(0.001,0.05,by=0.0005)
lasso_model<-train(y=yt,x=xt,method = 'lasso',
       trControl=control,tuneGrid =expand.grid(fraction=lambda_grid),
       preProcess=c('center', 'scale'))
lasso_model
predict(lasso_model$finalModel,type="coef",mode="fraction",s=0.05)
```

[6] `https://cran.r-project.org/web/packages/data.table/data.table.pdf`, p. 6

```
predict(lasso_model, test[1,])
```

The command `lasso_model` tells us which is the best λ. The first `predict(...)` above gives us the coefficient of the vector w. Some are zero, some are big. This teaches us which variables are important for the outlet sales. Finally, the second `predict(...)` makes a prediction of the outlet sales for the first observation of the test set.

2.7.2. Boston Housing data set. You can find a description of the data on `https://www.kaggle.com/c/boston-housing`. The file is named `BostonHousing.csv`. The goal is to predict `medv` (median-value of owner-occupied home) using a linear regression.

2.7.3. Summary of useful `R` commands. Read a `.csv` file:

```
data<-read.csv("data.csv")
```

(will produce a data-frame, exactly what we want).

Creat dummy variables

```
library(caret)
data_dummy<-dummyVars("~.", data, fullRank=True)
```

You can change the formula ‘‘~ .’’ (be careful and read the man page). The `fullRank=TRUE` is used to avoid redundancy in the variables.

Get indexes satisfying a condition:

```
num<-which(...)
```

Train a regression model (outcomes=`out`, incomes=`in`) (`library caret`)

```
model<-train(y=out, x=in, method='lasso', trControl=control, tuneGrid=grid)
```

where the `method` can be `'ridge'` (ridge regularization), `'lasso'` (for Lasso) and many others (see `http://topepo.github.io/caret/train-models-by-tag.html`), `trControl` can be something else (read `http://topepo.github.io/caret/using-your-own-model-in-train.html` and proceed with caution), `tuneGrid` is the grid on which your parameter is optimized. This is indeed a powerful method since, if you use for example the Lasso method, it will compute for you which is the best parameter.

2.8. Dummy variables

2.8.1. Introduction to dummy variables. Let us have a look into the `R` help (`help(dummyVars)`). The example given here is the following. We have a data-frame `when` (see Table 2.8.1) with 9 observations

	time	day
1	afternoon	Mon
2	night	Mon
3	afternoon	Mon
4	morning	Wed
5	morning	Wed
6	morning	Fri
7	morning	Sat
8	afternoon	Sat
9	afternoon	Fri

TABLE 2.8.1. Data-frame `when`

of 2 variables, the variable “time” has 2 levels and the variables “day” has 7 levels. We create dummy variables and show the new table with the following code.

```
library(caret)
mainEffects <- dummyVars(~ day + time, data = when,fullRank=True)
mainEffects
predict(mainEffects, when)
```

We get a data-frame similar to Table 2.8.2. It has to be understood in the following way. Observation

	day.Tue	day.Wed	day.Thu	day.Fri	day.Sat	day.Sun	time.afternoon	time.night
1	0	0	0	0	0	0	1	0
2	0	0	0	0	0	0	0	1
3	0	0	0	0	0	0	1	0
4	0	1	0	0	0	0	0	0
5	0	1	0	0	0	0	0	0
6	0	0	0	1	0	0	0	0
7	0	0	0	0	1	0	0	0
8	0	0	0	0	1	0	1	0
9	0	0	0	1	0	0	1	0

TABLE 2.8.2. Data-frame `predict(mainEffects,when)`

8 is ("Saturday","afternoon"). So on a scale from 0 to 1, how much Saturday-ish is it? The answer is 1 (see yellow cell in the table). And how much Tuesday-ish is it? The answer is 0 (see cyan cell in the table). What about observation 1? There is no column "day.Monday" in which we could put a 1 for this observation. But we see 0 in all columns from "day.Tue" to "day.Sun", so we know this observation should be 1 on the "day.Monday" scale. Here we have avoided having redundant information in our table by using the `fullRank=True` option above (try and see what happens with `fullRank=False`).

2.8.2. Exercise with dummy variables. Download the file `chap-02-princeton-salary.dat`. It looks like the Table in Figure (2.8.1) (sx= sex, rk = rank, yr = year in current rank, dg= degree, yd =

```
    sx   rk yr        dg yd    sl
  male full 25 doctorate 35 36350
  male full 13 doctorate 22 35350
  male full 10 doctorate 23 28200
female full  7 doctorate 27 26775
  male full 19   masters 30 33696
  male full 16 doctorate 21 28516
```

FIGURE 2.8.1. Princeton data

years since earning highest degree, sl = salary). Use the `dummyVars` command to create a data-frame containing the information of the original file, but only with numerical variables. See the answer in `chap-02-activity-2.8.2.R`.

CHAPTER 3

Logistic regression

3.1. Introduction

3.1.1. Outline of the plan. In the previous chapter, we have studied linear regression. This is a way of making predictions for the output when a new input is coming in. This is also a way to understand which features of the input are important in the answer/output. We are now going to use regression to perform binary classification. The idea is pretty simple, we have data points in $\mathbb{R}^D$ with labels that can be 0 or 1. What we can do is a regression: starting from the input points, we compute a linear combination of their features, this gives us a "score". We apply to this score the logistic function (already seen in Section 1.4.3, see Figure 1.4.5 for the shape of this function). This function transforms the score into a number in $[0,1]$. When a new inputs comes by, we apply to it this recipe and we get a number in $[0,1]$. We decide this number is our prediction probability of the new label to be 1. There is no reason that this prediction should be good if I compute the score with just any coefficients. The trick is that we choose the coefficients such that the predictions fit the actual labels as best as we can. This is called "logistic regression" and this is the subject of Section 3.2 below.

3.1.2. Credit card fraud detection. Credit card fraud is a huge concern for the finance and banking system. For example, fraud losses on UK-issued cards totaled £620.6 million in 2019 (source: `https://www.ukfinance.org.uk/system/files/Fraud-The-Facts-2020-FINAL-ONLINE-11-June.pdf`). Analyzing fraudulent transactions manually is unfeasible due to huge amounts of data and its complexity. However, given sufficiently informative features, one could expect it is possible to do using Machine Learning.

The data set. The datasets[1] contains transactions made by credit cards in September 2013 by European cardholders. This dataset presents transactions that occurred in two days, where we have 492 frauds out of 284,807 transactions. The dataset is highly unbalanced, the positive class (frauds) account for 0.172% of all transactions.

It contains only numerical input variables which are the result of a PCA transformation (see Section 1.3.2). Unfortunately, due to confidentiality issues, we cannot provide the original features and more background information about the data. Features V1, V2, ... V28 are the principal components obtained with PCA, the only features which have not been transformed with PCA are 'Time' and 'Amount'. Feature 'Time' contains the seconds elapsed between each transaction and the first transaction in the dataset. The feature 'Amount' is the transaction Amount, this feature can be used for example-dependent cost-sensitive learning. Feature 'Class' is the response variable and it takes value 1 in case of fraud and 0 otherwise.

Manipulation in R. We load the data.

```
library(data.table)
data<-read.csv('creditcard.csv')
```

We can have a look at the beginning of the data.

```
head(data)
```

This gives us something looking like Figure 3.1.1. The percentage of fraudulent activity is given by

```
ratio=sum(data$Class==1)/nrow(data)
print(ratio*100)
```

[1] `https://www.kaggle.com/mlg-ulb/creditcardfraud`

```
        V25        V26          V27         V28 Amount Class
 0.1285394 -0.1891148  0.133558377 -0.02105305 149.62     0
 0.1671704  0.1258945 -0.008983099  0.01472417   2.69     0
-0.3276418 -0.1390966 -0.055352794 -0.05975184 378.66     0
 0.6473760 -0.2219288  0.062722849  0.06145763 123.50     0
-0.2060096  0.5022922  0.219422230  0.21515315  69.99     0
-0.2327938  0.1059148  0.253844225  0.08108026   3.67     0
.....|
```

FIGURE 3.1.1. Credit card data

and it is 0.17%. So the data is extremely unbalanced. A predictor predicting 0 for any transaction will have an accuracy of 99,83%. So the accuracy is not a good indicator of a predictor performance. One could use the recall measure (also known as sensitivity). This is the ratio of true positives (TP) over the sum of TP and false negatives (FN). A true positive is a point which is predicted correctly to have label 1 and a false negative is a point that is predicted to have label 0 whereas it has, in reality, label 1. We write the formula for the recall indicator:

$$\text{Recall} = \frac{\text{TP}}{\text{TP}+\text{FN}}.$$

We will see the rest of the programming in Section 3.5.1.

3.2. Mathematical model

The model is the following. For an input $x = (x_1, \dots, x_D)^T$ in $\mathbb{R}^D$, we have an output y of law $\mathscr{B}(y|\text{sigm}(w^T x))$. $\mathscr{B}(\dots|p)$ is the Bernoulli law of parameter p (probability of having 1 is p and probability of having 0 is $1-p$). The vector $w = (w_1, \dots, w_D)^T$ is in $\mathbb{R}^D$. The function sigm is the sigmoid function and has been defined in Section 1.4.3. For $x = (x_1, \dots, x_D)^T$ in $\mathbb{R}^D$, we have written the transposition sign T to stress that x is a column vector. We write the law of y knowing x and w in the following way:

$$p(y|x,w) = \mathscr{B}(y|\text{sigm}(w^T x)). \tag{3.1}$$

If we want the parameter in the Bernoulli to be $w_0 + w^T x$, we can add a dimension by defining:

$$\overline{w} = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_D \end{pmatrix}, \overline{x} = \begin{pmatrix} 1 \\ x_1 \\ \vdots \\ x_D \end{pmatrix}.$$

Then the parameter can be written

$$w_0 + w^T x = \overline{w}^T \overline{x},$$

and we have again the same notation as in Equation (3.1) above.

3.2.1. Maximum Likelihood Estimator (MLE). You have observations $(x^{(i)}, y^{(i)})_{1 \le i \le N}$ ($y^{(i)}$ is in $\{0,1\}$). Suppose the parameter w is fixed. Density of $(y^{(1)}, \dots, y^{(N)})$ knowing $(x^{(1)}, \dots, x^{(N)})$ and w, with respect to the measure $\delta_0 + \delta_1$:

$$p(y^{(1)}, \dots, y^{(N)}|x^{(1)}, \dots, x^{(N)}, w) = \prod_{i=1}^{N} ((1 - \text{sigm}(w^T x^{(i)}))\, \mathbb{1}_{\{0\}}(y^{(i)}) + \text{sigm}(w^T x^{(i)})\, \mathbb{1}_{\{1\}}(y^{(i)}). \tag{3.2}$$

Simple explanation: density of $\mathscr{B}(p)$ is $x \mapsto p\,\mathbb{1}_{\{1\}}(x) + (1-p)\,\mathbb{1}_{\{0\}}(x)$. Indeed, if we compute, for any function f, ($X \sim \mathscr{B}(p)$)

$$\begin{aligned} \mathbb{E}(f(X)) &= p \times f(1) + (1-p) \times f(0) \\ &= \int_{\mathbb{R}} p\,\mathbb{1}_{\{1\}}(x) + (1-p)\,\mathbb{1}_{\{0\}}(x)(\delta_0(dx) + \delta_1(dx)). \end{aligned}$$

The $p(\dots)$ in Equation (3.2) above id called the likelihood. We now treat this likelihood as a function of w and we look for

$$\widehat{w} = \arg\max_w p(y^{(1)}, \dots, y^{(N)}|x^{(1)}, \dots, x^{(N)}, w).$$

This $\widehat{w}$ is called the Maximum Likelihood Estimator (MLE). It is a consistent estimator, meaning that it converges to the real w when N goes to infinity. The real w is the one linking the inputs and the outputs in Equation (3.1).

We take the $-\log$ of the likelihood above

$$\begin{aligned}\mathrm{NLL}(w) &= -\log(p(y^{(1)},\dots,y^{(N)}|x^{(1)},\dots,x^{(N)},w)) \\ &= -\sum_{i=1}^{N}\log(1-\mathrm{sigm}(w^T x^{(i)}))\mathbb{1}_{\{0\}}(y^{(i)}) + \log(\mathrm{sigm}(w^T x^{(i)}))\mathbb{1}_{\{1\}}(y^{(i)}) \\ &\qquad (\mu_i := \mathrm{sigm}(w^T x^{(i)})) = -\sum_{i=1}^{N}(1-y^{(i)})\log(1-\mu_i) + y^{(i)}\log(\mu_i)\,.\end{aligned}$$

The goal is now to find the minimum of NLL. For this we can use optimization algorithms. As we will see below, classical optimization algorithms use the gradient and the Hessian of NLL. So we compute these quantities here as an exercise.

EXERCISE 3.1. We define

$$X = \begin{bmatrix} x^{(1)T} \\ x^{(2)T} \\ \vdots \\ x^{(N)T} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & \dots & x_D^{(1)} \\ x_1^{(2)} & \dots & x_D^{(2)} \\ \vdots & \vdots & \vdots \\ x_1^{(N)} & \dots & x_D^{(N)} \end{bmatrix},$$

$$\mu = \begin{pmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_N \end{pmatrix}, y = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(N)} \end{pmatrix}.$$

Question: Show that the gradient of NLL in w is:

$$\nabla\mathrm{NLL}(w) = X^T(\mu - y)\,,$$

and show that the Hessian matrix of NLL in w is

$$H(w) = X^T S X \tag{3.3}$$

(where $S = \mathrm{diag}(\mu_i(1-\mu_i))_{1\le i\le N}$).

Answer: We have $\mathrm{NLL} : w \in \mathbb{R}^D \mapsto \mathbb{R}$. The gradient is

$$\nabla\mathrm{NLL}(w) = \begin{pmatrix} \vdots \\ \frac{\partial \mathrm{NLL}}{\partial w_j}(w) \\ \vdots \end{pmatrix}.$$

We derive each part of $\sum_{i=1}^{N}\dots$ (with respect to w_j for j fixed), using that $\mathrm{sigm}'(t) = \mathrm{sigm}(t)(1-\mathrm{sigm}(t))$,

$$\begin{aligned}\frac{\partial \log(\mu_i)}{\partial w_j} &= \frac{1}{\mu_i}\frac{\partial \mu_i}{\partial w_j} \\ &= \frac{1}{\mu_i}\frac{\partial (w^T x^{(i)})}{\partial w_j}\mathrm{sigm}(w^T x^{(i)})(1-\mathrm{sigm}(w^T x^{(i)}) \\ &= \frac{1}{\mu_i}\frac{\partial}{\partial w_j}(\sum_{j=1}^{D} w_j x_j^{(i)})\,\mathrm{sigm}(w^T x^{(i)})(1-\mathrm{sigm}(w^T x^{(i)}) \\ &= \frac{1}{\mu_i}x_j^{(i)}\,\mathrm{sigm}(w^T x^{(i)})(1-\mathrm{sigm}(w^T x^{(i)}) \\ &= \frac{\mathrm{sigm}(w^T x^{(i)})(1-\mathrm{sigm}(w^T x^{(i)})}{\mu_i}x_j^{(i)}\end{aligned}$$

$$\begin{aligned}\frac{\partial \log(1-\mu_i)}{\partial w_j} &= \frac{-1}{1-\mu_i}\frac{\partial \mu_i}{\partial w_j} \\ &= \frac{-\text{sigm}(w^T x^{(i)})(1-\text{sigm}(w^T x^{(i)})}{1-\mu_i} x_j^{(i)}\end{aligned}$$

$$\frac{\partial}{\partial w_j}\left(-\sum_{i=1}^{N}(1-y^{(i)})\log(1-\mu_i)+y^{(i)}\log(\mu_i)\right)$$
$$= -\sum_{i=1}^{N}\text{sigm}(w^T x^{(i)})(1-\text{sigm}(w^T x^{(i)})x_j^{(i)}(\frac{y^{(i)}}{\mu_i}-\frac{1-y^{(i)}}{1-\mu_i})$$

So

$$\begin{aligned}\nabla \text{NLL}(w) &= -\sum_{i=1}^{N}\text{sigm}(w^T x^{(i)})(1-\text{sigm}(w^T x^{(i)}))\left(\frac{y^{(i)}}{\mu_i}-\frac{1-y^{(i)}}{1-\mu_i}\right)x^{(i)} \\ &= -\sum_{i=1}^{N}\text{sigm}(w^T x^{(i)})(1-\text{sigm}(w^T x^{(i)}))\left(\frac{y^{(i)}(1-\mu_i)-\mu_i(1-y^{(i)})}{\mu_i(1-\mu_i)}\right)x^{(i)} \\ &= -\sum_{i=1}^{N}\mu_i(1-\mu_i)\times\frac{(y^{(i)}-\mu_i)}{\mu_i(1-\mu_i)}x^{(i)} \\ &= -\sum_{i=1}^{N}\left(y^{(i)}-\mu_i\right)x^{(i)}\end{aligned}$$

$$X = \begin{bmatrix} x^{(1)T} \\ x^{(2)T} \\ \vdots \\ x^{(N)T} \end{bmatrix} = \begin{bmatrix} x_1^{(1)} & \dots & x_D^{(1)} \\ x_1^{(2)} & \dots & x_D^{(2)} \\ \vdots & \vdots & \vdots \\ x_1^{(N)} & \dots & x_D^{(N)} \end{bmatrix}$$

This matrix X has N rows and D columns. We define vectors

$$x_j = \begin{pmatrix} x_j^{(1)} \\ x_j^{(2)} \\ \vdots \\ x_j^{(N)} \end{pmatrix}$$

(columns of the matrix X). We define

$$\mu = \begin{pmatrix} \mu_1 \\ \mu_2 \\ \vdots \\ \mu_N \end{pmatrix}, y = \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(N)} \end{pmatrix}.$$

We then have

$$\nabla \text{NLL}(w) = X^T(\mu - y) = \begin{pmatrix} \vdots \\ \frac{\partial \text{NLL}}{\partial w_j}(w) \\ \vdots \end{pmatrix}$$

We now want the Hessian matrix (we write the (j,k) term)

$$H(w) = \begin{bmatrix} \dots & \dots & \dots \\ \dots & \frac{\partial^2 \text{NLL}}{\partial w_j \partial w_k}(w) & \dots \\ \dots & \dots & \dots \end{bmatrix}.$$

Let us compute

$$\frac{\partial}{\partial w_k}\left(X^T\mu\right)_j = \frac{\partial}{\partial w_k}(x_j^{(1)}\mu_1 + \dots + x_j^{(D)}\mu_D)$$

$$\begin{aligned} &= \sum_{r=1}^{D} x_j^{(r)} \frac{\partial \mu_r}{\partial w_k} \\ &= \sum_{r=1}^{D} x_j^{(r)} \mu_r (1-\mu_r) x_k^{(r)} \\ &= \sum_{r=1}^{D} \mu_r (1-\mu_r) x_j^{(r)} x_k^{(r)} . \end{aligned}$$

So

$$H(w) = X^T S X$$

with $S = \mathrm{diag}(\mu_i(1-\mu_i))_{1\le i\le N}$.

REMARK 3.1. We have, for all i, $\mu_i \in (0,1)$. Suppose now that $N > D$. W have

$$X = [x_1, x_2, \dots, x_D]$$

and so (for z a column vector, $z = (z_1, \dots, z_D)^T$)

$$Xz = z_1 x_1 + \cdots + z_D x_D .$$

We compute

$$\begin{aligned} z^T X^T S X z &= \sum_{r=1}^{N} \mu_r (1-\mu_r) ((Xz)_r)^2 \\ &= \sum_{r=1}^{N} \mu_r (1-\mu_r) \left(\sum_{i=1}^{D} z_i x_i^{(r)} \right)^2 \end{aligned}$$

It is always ≥ 0. So the quadratic form associated to $H(w)$ is always nonnegative, which means that any critical point (i.e. a point where the gradient is zero) will be a minimum. If $z^T X^T S X z$ is 0, then, for all r, $\left(\sum_{i=1}^{D} z_i x_i^{(r)}\right)^2 = 0$, $\sum_{i=1}^{D} z_i x_i^{(r)} = 0$ so

$$\sum_{i=1}^{D} z_i x_i = 0$$

Suppose the rank of X is D (easy to check numerically), then the above equality implies $z_i = 0$ for all i. And so the matrix $H(w)$ is positive.

3.2.2. Optimization algorithms (non-exhaustive list).

Steepest descent (/gradient descent). We look for the minimum of a function $f : \mathbb{R}^D \to \mathbb{R}$. We do not assume much about this function and we all its gradient g. We define a sequence by recurrence

$$\begin{cases} \theta_0 \in \mathbb{R}^D , \\ \theta_{k+1} = \theta_k - \eta_k g(\theta_k) , \end{cases}$$

where the step-size (/learning rate) $(\eta_k)_{k\ge 0}$ has to be chosen wisely. If the steps η_k are too small, the convergence will be slow, and if they are too big, the sequence $(\theta_k)_{k\ge 0}$ might never converge).

We observe that $f(\theta + \eta d) \approx f(\theta) + \eta \langle g(\theta), d\rangle = f(\theta) + \eta g(\theta)^T d$[2] (this is a Taylor expansion to the first order, where d is our descent direction). If we take $d = -\eta g(\theta)$ then $\eta\langle g(\theta), d\rangle \le 0$. So, if η is small enough, we can guarantee that $f(\theta + \eta d) \le f(\theta)$.

But we can do better. We can pick η in order to minimize: $\phi(\eta) = f(\theta + \eta d)$. This is called line minimization (/line search) (see Figure 3.2.1 for an illustration). This, in turn, is a one dimensional problem which can be solved using various methods. In Figure 3.2.1, we have drawn the level lines of the function f (the minimum is supposed to be in the center of the drawing). We start with θ_0 at the "start" point. We perform our line search by moving on the line having the direction of the gradient at the "start" point. Observe that the line search will stop somewhere where the line is perpendicular to the local gradient. This is why the trajectory might exhibit a zig-zag behavior.

[2] $\langle \dots, \dots \rangle$ is the usual scalar product

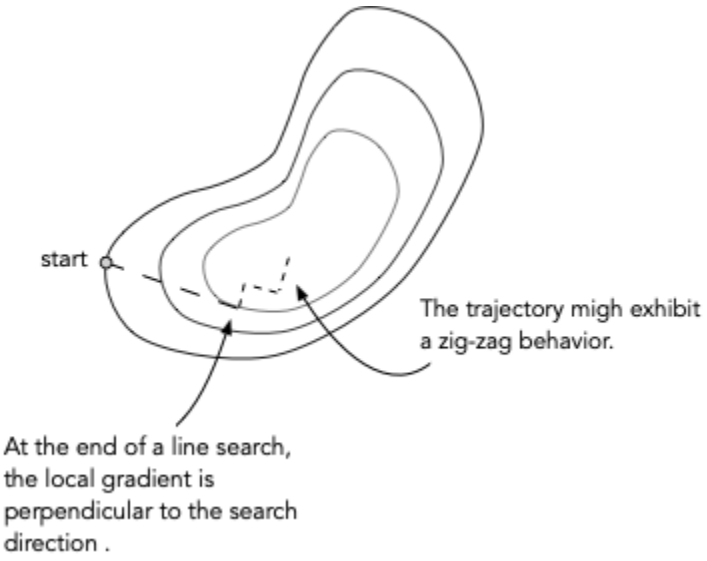


FIGURE 3.2.1. Line search

Newton's method (for a strictly convex function). We could obtain a faster optimization method bu taking into account the curvature of the space (i.e. the Hessian). We get what are called second order optimization methods. The primary example is Newton's method.

We look for the minimum of a function $f : \mathbb{R}^D \to \mathbb{R}$, with gradient ∇f and Hessian matrix H. We define a sequence by recurrence

$$\begin{cases} \theta_0 \in \mathbb{R}^D , \\ \theta_{k+1} = \theta_k - \eta_k H_k^{-1} g_k , \end{cases}$$

where

$$\begin{cases} H_k = H(\theta_k) , \\ g_k = \nabla f(\theta_k) , \\ d_k \text{ solution of } H_k d_k = -g_k \\ \eta_k = \arg\min_\eta f(\theta_k + \eta_k d_k) . \end{cases}$$

The idea behind this algorithm is the following. Let us write a Taylor expansion (around θ_k):

$$f(\theta) = f(\theta_k) + g_k^T(\theta - \theta_k) + \frac{1}{2}(\theta - \theta_k)^T H_k(\theta - \theta_k) + \dots . \tag{3.4}$$

This can be rewritten as (with $b = g_k - H_k\theta_k$, $c = f(\theta_k) - H_k\theta_k$, $A = H_k/2$)

$$f(\theta) = c + b^T\theta + \theta^T A\theta + \dots .$$

This last expression is minimal at

$$\theta = -\frac{1}{2}A^{-1}b = \theta_k - H_k^{-1}g_k .$$

Thus the Newton step $d_k = -H_k^{-1}g_k$ is what should be added to θ_k to minimize the second order approximation of f around θ_k. We then choose η_k using a line search to be even more optimal.

Application of Newton's method to logistic regression. We want to apply the above method to find the MLE for the binary logistic regression. We fix $\eta_k = 1$ ($\forall k$). At iteration k, we set $\mu_{i,k} = \text{sigm}(\theta_k^T x_i)$,

$$\overline{\mu}_k = \begin{pmatrix} \mu_{1,k} \\ \vdots \\ \mu_{N,k} \end{pmatrix},$$

$S_k = \text{diag}(\mu_{i,k}(1 - \mu_{i,k}))_{1 \le i \le N}$. We have the recursion formula:

$$\begin{aligned} \theta_{k+1} &= \theta_k - H_k^{-1}g_k \\ &= \theta_k + (X^T S_k X)^{-1} X^T (y - \overline{\mu}_k) \end{aligned}$$

$$\begin{aligned} &= (X^T S_k X)^{-1}((X^T S_k X)\theta_k + X^T(y-\overline{\mu}_k)) \\ &= (X^T S_k X)^{-1} X^T (S_k X\theta_k + y - \overline{\mu}_k) \\ &= (X^T S_k X)^{-1} X^T S_k z_k, \end{aligned}$$

with $z_k = X\theta_k + S_k^{-1}(y-\overline{\mu}_k)$ (called the working response). We write $z = (z_{1,k}, z_{2,k}, \ldots, z_{N,k})^T$. We set D_k such that $S_k = D_k^2$. Because of Equation (2.2), the θ minimizing $(D_k z_k - D_k X\theta)^T(D_k z_k - D_k X\theta)$ is

$$\theta = (X^T D_k D_k X)^{-1} X^T D_k D_k z_k = (X^T S_k X)^{-1} X^T S_k z_k.$$

And so θ_{k+1} is a minimizer of

$$\theta \mapsto (D_k z_k - D_k X\theta)^T(D_k z_k - D_k X\theta) = \sum_{i=1}^{N} \sqrt{\mu_{i,k}(1-\mu_{i,k})}(z_{i,k} - \theta^T x_i)^2$$

(which is called a weighted least square problem). We can write, for each component $z_{i,k}$:

$$z_{i,k} = \theta_k^T x_i + \frac{y_i - \mu_{i,k}}{\mu_{i,k}(1-\mu_{i,k})}.$$

This algorithm is known as iterated re-weighted least square (IRLS) since at each iteration, we solve a weighted least squares problem, where the weight matrix changes at each iteration.

3.3. l_2-regularization (for logistic regression)

If the data is such that all $x^{(i)}$ with $y^{(i)} = 0$ are smaller than all $x^{(j)}$ with $y^{(j)} = 1$, we say that the data is linearly separable. In this case the MLE is obtained when $\|w\| \to +\infty$. This situation corresponds to an infinitely steep sigmoid function (of the kind $x \mapsto \mathbb{1}_{w^T x > m_0}$), also known as linear threshold limit. This assigns the maximal amount of probability mass to the training data and this will not be good for future predictions. This is clearly a case of over-fitting.

To prevent this, we can use a l_2-regularization (similar to the ridge regression of Section 2.4). The new function we want to minimize (also called objective function), gradient and Hessian are given by

$$\begin{cases} \widetilde{f}(w) &= \text{NLL}(w) + \lambda w^T w, \\ \widetilde{g}(w) &= g(w) + \lambda w, \\ \widetilde{H}(w) &= H(w) + \lambda \text{Id}_D. \end{cases}$$

The parameter λ is user's choice. As in the case of the ridge regression, the penalization term will prevent the solution of the optimization problem to be too big. Again, we can apply the IRLS algorithm of Section 3.2.2 to the objective function $\widetilde{f}$.

3.4. Advantages/disadvantage

Advantages.

- Logistic regression performs well when the data is linearly separable.
- Logistic regression is not prone to over-fitting and when it does, you can consider l_2-regularization.
- Logistic regression is easy to implement and to train.

Disadvantages.

- The main limitation is the assumption of linearity between the input variables and the output. There is no reason, in real life, that this should be the case.
- If the number of observations are lesser than the number of features, Logistic Regression should not be used, otherwise it may lead to overfit.

3.5. Examples

3.5.1. Detecting credit card fraudulent activity (continuation of Section 3.1.2). We start by splitting our data into a training set and a test set.

```
library(data.table)
library(ggplot2)
library(plyr)
library(dplyr)
library(corrplot)
library(pROC)
library(glmnet)
library(caret)
library(Rtsne)
library(xgboost)
library(doMC)
set.seed(42)
data_Class <- as.numeric(data$Class)
train_index <- createDataPartition(data$Class, times = 1, p = 0.8, list = F)
X_train <- data[train_index,]
X_test <- data[-train_index,]
y_train <- data_Class[train_index]
y_test <- data_Class[-train_index]
```

The command we use here (`createDataPartition`) is such that the percentage of fraudulent transactions is roughly the same in the training set and in the test set.

We carry on with the credit card example of the introduction. We call the regression.

```
log_mod <- glm(Class ~ ., family = "binomial", data = X_train)
summary(log_mod)
```

We then use a threshold of 0.5 to transform probability predictions into binary variables.

```
tableau=c(as.numeric(predict(log_mod,X_test,type='response')>0.5))
tableau<-factor(tableau)
y_test<-factor(y_test)
conf_mat<-confusionMatrix(y_test,tableau)
print(conf_mat)
```

We see that our recall is $55/(55+7) = 88.7\%$, which is pretty good. The confusion matrix tells us that a lot of fraudulent transactions go undetected and that the false negatives are scarce.

3.5.2. Credit card default (activity). We use here the file[3] `chap03-credit-card-default.csv`. This dataset contains information on default payments, demographic factors, credit data, history of payment, and bill statements of credit card clients in Taiwan from April 2005 to September 2005. The data set could be used to estimate the probability of default payment by credit card client using the data provided. These attributes are related to various details about a customer, his past payment information and bill statements. In the data, the last column is 1 of there is a default on the credit card in the month following the gathering of the data. Imagine you are a data scientist employed by a bank. It would be very valuable to be able to predict if a customer is likely to default on his credit card during the next month, particularly when deciding to grant a loan to this customer. Such knowledge can potentially save the bank a lot of money.

Here are the definition of each variables:

- ID : ID of each client
- LIMIT_BAL : Amount of given credit in NT dollars (includes individual and family/supplementary credit
- SEX : Gender (1=male, 2=female)
- EDUCATION : (1=graduate school, 2=university, 3=high school, 4=others, 5=unknown, 6=unknown)

[3] `https://www.kaggle.com/uciml/default-of-credit-card-clients-dataset`

- MARRIAGE : Marital status (1=married, 2=single, 3=others)
- AGE : Age in years
- PAY_0 : Repayment status in September, 2005 (-1=pay duly, 1=payment delay for one month, 2=payment delay for two months, ... 8=payment delay for eight months, 9=payment delay for nine months and above)
- PAY_2 : Repayment status in August, 2005 (scale same as above)
- PAY_3 : Repayment status in July, 2005 (scale same as above)
- PAY_4 : Repayment status in June, 2005 (scale same as above)
- PAY_5 : Repayment status in May, 2005 (scale same as above)
- PAY_6 : Repayment status in April, 2005 (scale same as above)
- BILL_AMT1 : Amount of bill statement in September, 2005 (NT dollar) - BILL_AMT2 : Amount of bill statement in August, 2005 (NT dollar)
- BILL_AMT3 : Amount of bill statement in July, 2005 (NT dollar)
- BILL_AMT4 : Amount of bill statement in June, 2005 (NT dollar)
- BILL_AMT5 : Amount of bill statement in May, 2005 (NT dollar)
- BILL_AMT6 : Amount of bill statement in April, 2005 (NT dollar)
- PAY_AMT1 : Amount of previous payment in September, 2005 (NT dollar)
- PAY_AMT2 : Amount of previous payment in August, 2005 (NT dollar)
- PAY_AMT3 : Amount of previous payment in July, 2005 (NT dollar)
- PAY_AMT4 : Amount of previous payment in June, 2005 (NT dollar)
- PAY_AMT5 : Amount of previous payment in May, 2005 (NT dollar)
- PAY_AMT6 : Amount of previous payment in April, 2005 (NT dollar) - default.payment.next.month : Default payment (1=yes, 0=no)

Try a logistic regression on this data set.

3.5.3. Summary of useful `R` commands. Create a data partition in which the features of `data$class` are evenly distributed between the training set and the test set (the training set taking 80% of the data in this example, `list=F` because we want our resulting indexes to be in a matrix and not a list).

```
library(caret)
train_index <- createDataPartition( data$Class , times = 1 , p =0.8, list=F)
train_set<-data[train_index,]
test_set<-data[-train_index,]
```

Linear regression (on a data-frame `dframe` in which `dframe$Y` is the response variable (a.k.a. output)

```
library(stats)
lin_regression<-glm(Y~.,data=dframe)
```

If we want a logistic regression, we use

```
lin_regression<-glm(Y~.,family='binomial',data=dframe)
```

(this applies the sigmoid function to the result of the regression).

Predict a probability from a logistic regression model:

```
predict(lin_regression,newdata=...,type='response')
```

(the option `type='response'` is here to say we want a predicted probability).

CHAPTER 4

Support Vector Machine (SVM)

4.1. Introduction

In Section 4.2 below, we will study another linear problem (regression is a linear problem). We have data points in $\mathbb{R}^D$ with labels that can be -1 or 1. We suppose we can draw a hyperplane between the points labeled -1 and the points labeled $+1$ (in which case we say that the points are linearly separable). Afterwards, any new incoming point will be labeled according to which side of the hyperplane it happens to be.

But what can be done when the points are not linearly separable? The Section on non-linear SVM (Section 4.3) gives an answer using a very powerful idea, which we will described fully in the said Section.

4.2. Linear support vector machine (SVM)

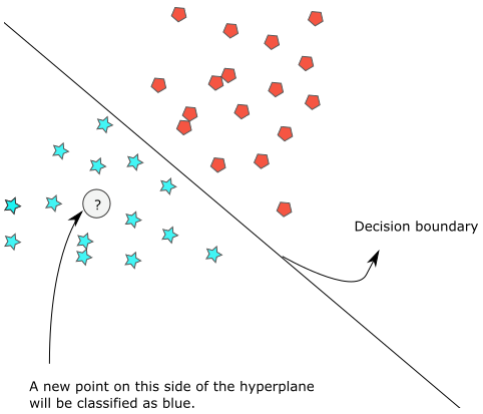


FIGURE 4.2.1. Hyperplane decision boundary

The linear SVM is a classifier that uses a hyperplane as a decision boundary. Imagine you have a data set $(x^{(i)}, t^{(i)})_{1\leq i\leq N}$ with the points $x^{(i)}$ in $\mathbb{R}^D$ and the labels $t^{(i)}$ in $\{-1,1\}$. In Figure 4.2.1, we colored the points with label equal to 1 in red and the other points in blue. Remember that a hyperplane of $\mathbb{R}^D$ is an affine sub-space of dimension $D-1$. We suppose the data is linearly separable, that is there exists a hyperplane H such that the red points are on one side of the hyperplane and the blue points are on the other side. We can write H as $H = \{x \in \mathbb{R}^D : w^T x + b = 0\}$ for some $w \in \mathbb{R}^D$ and some $b \in \mathbb{R}$. In Figure 4.2.1, the dimension D equals 2 so the hyperplane H is a line. The hyperplane H is called the decision boundary. When we have a new point x_{new} coming in, we predict its label t_{new} by $t_{\text{new}} = \text{sign}(w^T x_{\text{new}} + b)$ ($t_{\text{new}} \in \{-1,+1\}$). So, from the hyperplane H, we build a decision function:

$$x_{\text{new}} \mapsto t_{\text{new}} = \text{sign}(w^T t_{\text{new}} + b)\,.$$

In the unlikely case that $w^T t_{\text{new}} + b = 0$, we decide arbitrarily that $\text{sign}(w^T t_{\text{new}} + b) = +1$.

4.2.1. The margin. The margin is defined as the perpendicular distance from the decision boundary to the closest point on either side.We always choose a hyperplane such that the distances to the closest points on each side are the same. In Figure 4.2.2, the distance from the hyperplane to the closest

point is γ, we have one red point ($x^{(2)}$) at distance γ from the hyperplane and one blue point ($x^{(1)}$) at distance γ from the hyperplane. In this figure, the margin is γ.

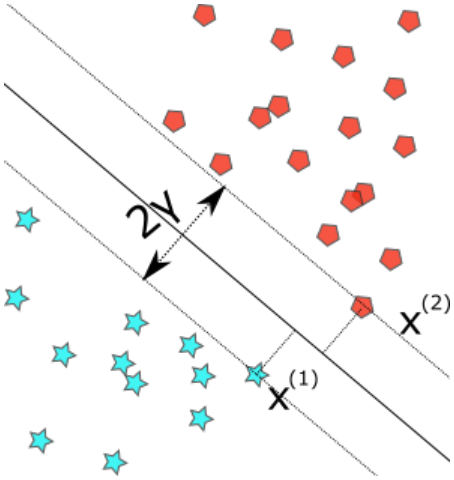


FIGURE 4.2.2. Margin

4.2.2. Maximizing the margin. We decide we want a hyperplane separating the data (blue points on one side and red points on the other side) and such that the margin γ is as big as possible. Why

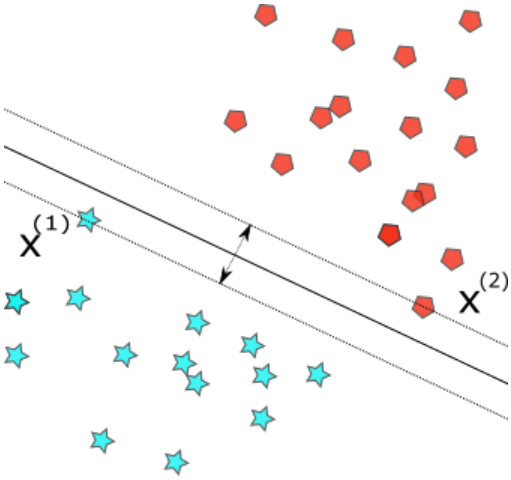


FIGURE 4.2.3. Small margin

is that? Let us have a look at Figure 4.2.3. We use here the same points as in Figure 4.2.2. We draw a hyperplane separating the two sets (blue stars en red polygons). This one has a small margin but we are happy as long as it separates correctly the two sets. Now imagine, we need to classify a new point (indicated with an arrow in Figure 4.2.4). With our new hyperplane, it will be classified as blue, whereas it is more likely to be red. Observe that it is classified as red by the previous hyperplane, which we have drawn in the same picture. Having this type of case in mind, we always want to maximize the margin.

The vector w (appearing in the equation of the hyperplane H) is perpendicular to H so (with the points $x^{(1)}$, $x^{(2)}$ from the previous Section)

$$2\gamma = \pm \frac{1}{\|w\|} w^T (x^{(1)} - x^{(2)}).$$

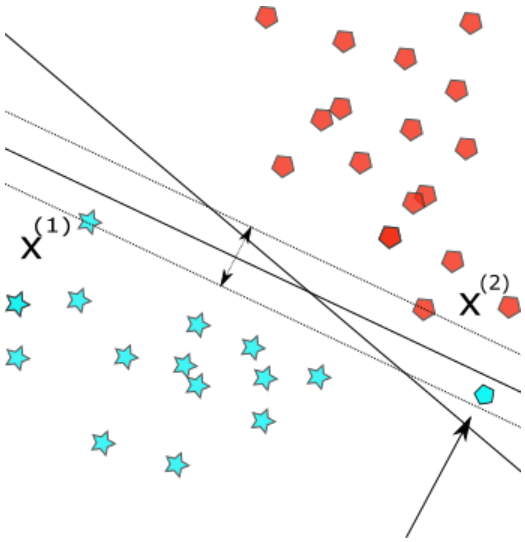


FIGURE 4.2.4. Wrong classification

If necessary, we replace w by $-w$, in order to have

$$2\gamma = +\frac{1}{\|w\|} w^T (x^{(1)} - x^{(2)}).$$

The decision function $x_{\text{new}} \mapsto t_{\text{new}} = \text{sign}(w^T x_{\text{new}} + b)$ is invariant by scaling of its argument by a positive constant, This means that we can multiply $w^T x_{new} + b$ by a positive λ, this will leave the output t_{new} unchanged. Therefore, we can fix the scaling of w and b such that $w^T x + b = \pm 1$ for $x \in \{x^{(1)}, x^{(2)}\}$. Thus

$$\begin{aligned} 2\gamma &= \frac{w^T}{\|w\|}(x^{(1)} - x^{(2)}) \\ &= \frac{1}{\|w\|}(w^T x^{(1)} + b - w^T x^{(2)} - b) \\ &= \frac{2}{\|w\|}. \end{aligned}$$

And so $\gamma = 1/\|w\|$.

So, in order to maximize the margin, we have to minimize $\|w\|$. This has to be done under the following constrains:

$$\begin{cases} w^T x + b \geq 1 & \text{for all red } x, \\ w^T x + b \leq -1 & \text{for all blue } x. \end{cases} \tag{4.1}$$

Optimization task. We formalize the problem in the following form:

$$\begin{cases} \text{find} & \widehat{w} = \arg\min_w \|w\|^2 \\ \text{subject to} & t^{(n)}(w^T x^{(n)} + b) \geq 1, \forall n. \end{cases} \tag{4.2}$$

To solve this optimization problem with constrains, we need to incorporate the constrains into the objective function through a set of Lagrange multipliers (see Section 10.1 in the Appendix for details on Lagrange multipliers, or [**Bin01**]). Our new objective function is

$$\begin{cases} (\widehat{w}, \widehat{\alpha}) = \arg\min_{w,\alpha} \frac{1}{2} w^T w - \sum_{n=1}^N \alpha_n (t^{(n)}(w^T x^{(n)} + b) - 1) \\ \text{subject to } \alpha_n \geq 0 \text{ for all } n. \end{cases} \tag{4.3}$$

The idea of the Lagrange multipliers technique is that a solution to Equation (4.2) is a solution to Equation (4.3), and that the reverse implication is true (without proof).

We look for a critical point of the new objective function ($f_L(w,b,\alpha) = \frac{1}{2}w^T w - \sum_{n=1}^N \alpha_n (t^{(n)}(w^T x^{(n)} + b) - 1)$)

$$\begin{cases} \nabla_w f_L &= w - \sum_{n=1}^N \alpha_n t^{(n)} x^{(n)} \\ \frac{\partial f_L}{\partial b} &= -\sum_{n=1}^N \alpha_n t^{(n)}. \end{cases}$$

At a critical point:

$$\begin{cases} w &= \sum_{n=1}^N \alpha_n t^{(n)} x^{(n)} \\ \sum_{n=1}^N \alpha_n t^{(n)} &= 0. \end{cases}$$

If we compute the value of f_L at a critical point, we get:

$$\frac{1}{2}w^T w - \sum_{n=1}^N \alpha_n (t^{(n)}(w^T x^{(n)} + b) - 1) =$$

$$\frac{1}{2}\left(\sum_{n=1}^N \alpha_n t^{(n)} x^{(n)}\right)^T \left(\sum_{n=1}^N \alpha_n t^{(n)} x^{(n)}\right) - \sum_{n=1}^N \alpha_n \left(t^{(n)}\left(\sum_{n=1}^N \alpha_m t^{(m)} x^{(m)T} x^{(n)} + b\right) - 1\right) =$$

$$\frac{1}{2}\sum_{n,m=1}^N \alpha_n \alpha_m t^{(n)} t^{(m)} x^{(m)T} x^{(n)} - \sum_{n,m=1}^N \alpha_n \alpha_m t^{(n)} t^{(m)} x^{(m)T} x^{(n)} - \sum_{n=1}^N \alpha_n t^{(n)} b + \sum_{n-1}^N \alpha_n =$$

$$(\text{as } \sum_{n=1}^N \alpha_n t^{(n)} = 0)$$

$$\sum_{n=1}^N \alpha_n - \frac{1}{2}\sum_{n,m=1}^N \alpha_n \alpha_m t^{(n)} t^{(m)} x^{(m)T} x^{(n)}.$$

There is a continuum of such critical points. We want to find the one minimizing the above expression, that is

$$\begin{cases} \widehat{\alpha} = \arg\min_\alpha \sum_{n=1}^M \alpha_n - \frac{1}{2}\sum_{n,m=1}^N \alpha_n \alpha_m t^{(n)} t^{(m)} x^{(m)T} x^{(n)} \\ \text{under } \alpha_n \geq 0 \ (\forall n),\ \sum_{n=1}^N \alpha_n t^{(n)} = 0. \end{cases}$$

This is known as the dual optimization problem. There is no explicit solution for $\widehat{\alpha}$ but, as the objective function is quadratic, this new problem is relatively easy to solve numerically.

Making predictions. Once, we have the α_n, we need b to be able to make predictions. We use the fact that if $x^{(n)}$ is one of the closest point to the decision boundary: $t_n(w^T x^{(n)} + b) = 1$. Also, we have that: $t_n = 1/t_n$, $w = \sum_{m=1}^N \alpha_m t^{(m)} x^{(m)}$, so we get

$$b = t^{(n)} - \sum_{m=1}^N \alpha_m t^{(m)} x^{(m)T} x^{(n)}. \tag{4.4}$$

When a new point x_{new} comes by, we can make the following prediction:

$$t_{\text{new}} = \text{sign}\left(\sum_{m=1}^N \alpha_m t^{(m)} x^{(m)T} x_{\text{new}} + b\right). \tag{4.5}$$

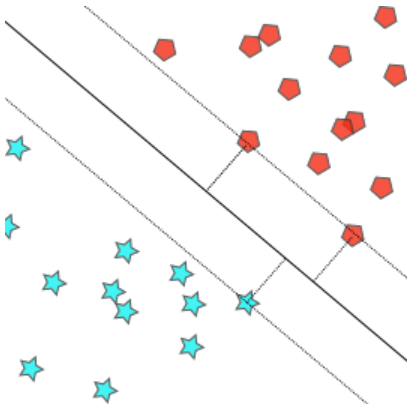


FIGURE 4.2.5. Support vectors

4.2.3. Support vectors. The set of points closest to the decision boundary are known collectively as the support vectors. On Figure 4.2.5, we have three such points. Suppose we are given the support vectors, the decision boundary is a hyperplane between them such that the margin is maximal. This decision boundary is thus uniquely determined by the support vectors. So the decision function depends only on a small subset of the data. We say we have a sparse solution. Consider classifying a test point using KNN when the training data consists of several thousand objects ... it could take a lot of time.

There is a drawback to this sparsity: moving a single point in the data might have a huge effect on the boundary. Another bad point is that this method is implementable only if there is a hyperplane separating the two populations in the data. This is called a hard margin SVM.

4.2.4. Soft margins. We want to allow points to potentially lie on the wrong side of the boundary. To achieve this, the constrain becomes

$$t^{(n)}(w^T x^{(n)} + b) \geq 1 - \xi_n \, (0 \leq \xi_n \leq 1)$$

(we give the system some "slack" in the form of the ξ_n). The new optimization task is

$$\text{(4.6)} \qquad \begin{cases} \text{find } \arg\min_w \frac{1}{2} w^T w + C \sum_{n=1}^N \xi_n \\ \text{subject to } \xi_n \geq 0 \text{ and } t^{(n)}(w^T x^{(n)} + b) \geq 1 - \xi_n \ (\forall \text{n}) \, . \end{cases}$$

The new parameter C controls to what extent we are willing to allow points to sit within the margin band on the wrong side of the decision boundary. A typical solution can be seen in Figure 4.2.6, the points with a non zero parameter $\xi_{...}$ are circled in green. Once again, we see a term $C \sum_{n=1}^N \xi_n$ that

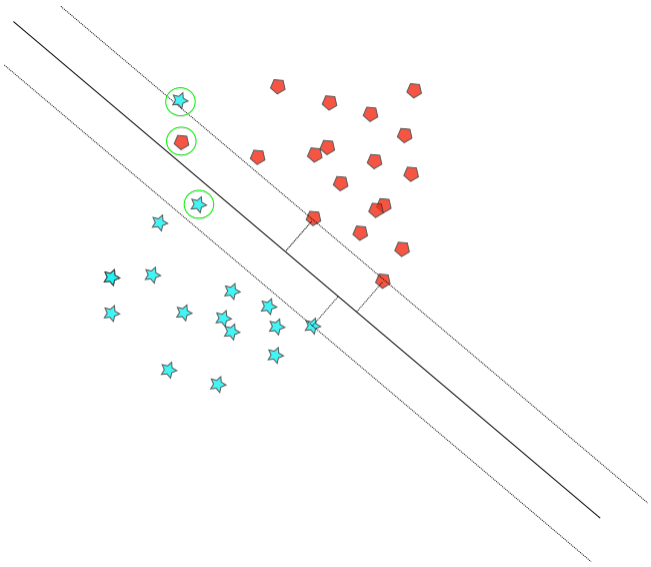


FIGURE 4.2.6

can be understood as a penalization term. The constant C needs to be fixed. We can set this using cross-validation.

We now need to find the maximum of the following quadratic problem

$$\begin{cases} \arg\max_\alpha \sum_{n=1}^N \alpha_n - \frac{1}{2} \sum_{n,m} \alpha_n \alpha_m t^{(n)} t^{(m)} x^{(n)T} x^{(m)} \\ \text{subject to } \sum_{n=1}^N \alpha_n t^{(n)} = 0 \text{ and } 0 \leq \alpha_n \leq C \ (\forall n). \end{cases}$$

To compute b, we need to find the support vector with the highest value of $w^T x^{(n)}$ and compute b from Equation (4.4).

4.2.5. Advantages/disadvantages.

Advantages:

- Memory efficiency. You need only to store the support vector in memory when making decisions (example of a problem in high dimension : document classification)
- High dimensionality. SVM is particularly suited for problems in high dimension

Disadvantages:

- Non-probabilistic: There is no probabilistic interpretation (you are either in a group or not). However, one could measure the effectiveness of the classification by the distance to the boundary.
- It will work only of the data is linearly separable (or nearly so if we use the soft margin version).
- The training phase can take a long time if the data set is big.
- There is no easy interpretation of what the decision boundary means.

4.3. Kernelized SVM

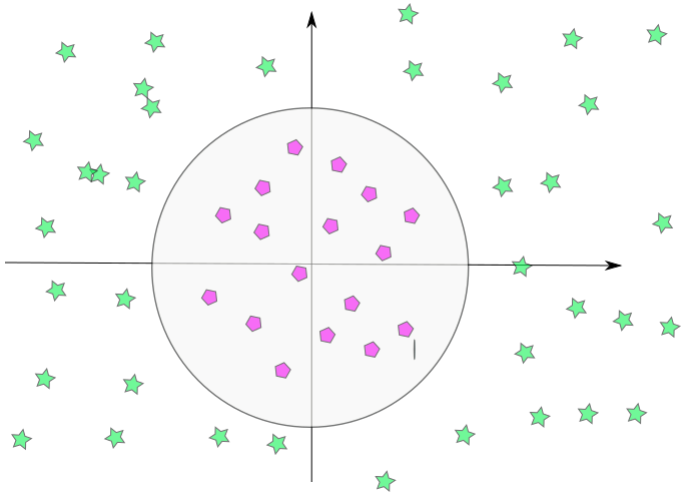


FIGURE 4.3.1. Non linearly separable data

4.3.1. Description. Let us have a look at Figure 4.3.1. We have data points in $\mathbb{R}^2$, some of them purple (polygons) and some of them green (stars). The two family of points cannot be separated by a straight line. Even, if we allow some points to lie on the wrong side of the decision boundary, this will not be efficient to look for a hyperplane separating the data. However, if we take the images of the points by the transformation $f : (x_1, x_2) \in \mathbb{R}^2 \mapsto x_1^2 + x_2^2 \in \mathbb{R}$, we obtain something similar to Figure 4.3.2. After

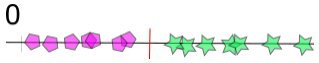


FIGURE 4.3.2. After transformation

the transformation, the two families are linearly separable. As they lie in $\mathbb{R}$, we mean here that there is a value t_0 in $\mathbb{R}$ such that the green points are on one side and the purple points on the other side. On our drawing, such a value t_0 is represented by a small red line.

So here is one idea, we take a transformation ϕ from our original space $\mathbb{R}^D$ to some other space $\mathbb{R}^p$. If we obtain points that are linearly separable (or nearly so, if we allow points on the wrong side of the boundary) then, we can use our linear SVM. For any new point x^{new}, we look at $\phi(x^{\text{new}})$ and check on which side of the boundary it lies.

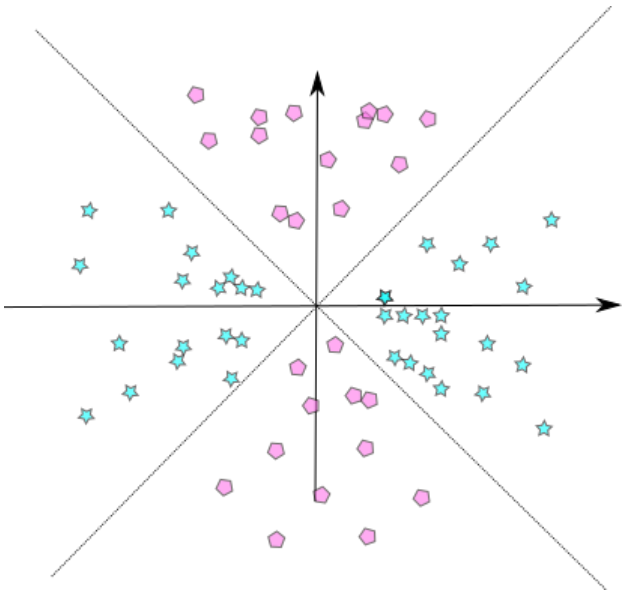


FIGURE 4.3.3. Another non-linearly separable data

EXERCISE 4.1. Let is have a look at Figure 4.3.3. What transformation can transform the two sets of points (blue stars and pink pentagons) into two linearly separable sets?

Answer: For any (x,y) in $\mathbb{R}^2$ can be represented by its polar coordinates (r,θ) ($r \in \mathbb{R}^+$, $r = \sqrt{x^2+y^2}$, $\theta \in [0,2\pi]$ such that $\cos(\theta) = x/r$, $\sin(\theta) = y/r$). One possible transformation is $(x,y) \mapsto |\sin(\theta)|$. The pink pentagons will end up in $(1/\sqrt{2},1]$ ($1/\sqrt{2}$ in the sin of $\pi/4$) and the blue stars will end up in $[0,1/\sqrt{2})$.

But there is something even more clever to do. When we look at Equations (4.1), (4.5), (4.6), we see that when we do the classification, we need to be able to compute scalar products in $\mathbb{R}^D$ (with column vectors x and x', the scalar product can be written $\langle x,x'\rangle = x^T x'$). So if we want to do a prediction in our new space $\mathbb{R}^p$, we need to be able to compute scalar product of the form $\phi(x^{(n)})^T\phi(x^{\text{new}})$ or $\phi(w)^T\phi(x^{(n)})$, where $x^{(n)}$ is one of the data points and w is the vector defining the hyperplane. So we do not need to think in term of transformation anymore. Instead, if we can show that some function k is such that $k(x,x') = \phi(x)^T\phi(x')$ for some transformation ϕ, we are free to use $k(\dots)$ in place of any inner product. Functions k that corresponds to an inner product as above (in some space) are called kernel functions.

Our optimization task and decision function (in the soft margin version) are rewritten to include kernel functions and we get the optimization task

$$\begin{cases} \arg\max_\alpha \sum_{n=1}^N \alpha_n - \frac{1}{2}\sum_{m,n=1}^N \alpha_n\alpha_m t_n t_m k(x_n,x_m) \\ \quad \text{subject to } \sum_{n=1}^N \alpha_n t_n = 0 \text{ and } 0 \le \alpha_n \le C,\ \forall n \end{cases}$$

In the original SVM, we could compute the decision boundary exactly as it consisted of values of x that satisfied $w^T x + b = 0$. We can no longer compute w as it would be given by $\sum_{n=1}^N \alpha_n t^{(n)}\phi(x^{(n)})$ and we do not necessarily know $\phi(x^{(n)})$ (we only know $k(x^{(n)},x^{(m)}) = \phi(x^{(n)})^T\phi(x^{(m)})$ for various n, m). Therefore, in order to draw the decision boundary, we have to evaluate $\sum_{n=1}^N \alpha_n t^{(n)} k(x^{(n)},x^{(\text{new})})$over a grid of values of $x^{(\text{new})}$ and then draw the contour corresponding to $\sum_{n=1}^N \alpha_n t^{(n)} k(x^{(n)},x^{(\text{new})}) = 0$.

4.3.2. Most popular kernels.

- Linear: $k(x,x') = x^T x'$.
- Gaussian (also known as Radial Basis Function or RBF): $k(x,x') = \exp\{-\gamma(x-x')^T(x-x')\}$.
- Polynomial: $k(x,x') = (1+x^T x')^\gamma$.

Linear is what we used in Section 4.2 The Gaussian and polynomial kernel have a parameter γ that must be tuned by the user (usually by cross-validation).

4.3.3. Tuning γ. Modifying γ changes the *implicit* transformation ϕ, which will in turn change the decision boundary. For the Gaussian kernel, increasing γ has the effect of increasing the complexity of the boundary in the original space. For this Gaussian kernel, a γ "too big" is such that there are many support vectors and the solution can no longer considered to be sparse.

4.3.4. Kernelization. The SVM is not the only algorithm that can be kernelized.

EXAMPLE 4.1. The KNN classifier requires the computation of the distances between $x^{(\text{new})}$ and each $x^{(n)}$. The square of this distance can be expressed as

$$(x^{(\text{new})} - x^{(n)})^T (x^{(\text{new})} - x^{(n)}) = (x^{(\text{new})})^T x^{(\text{new})} + (x^{(n)})^T x^{(n)} - 2(x^{(\text{new})})^T x^{(n)}.$$

We can replace this by a kernelized version:

$$k(x^{(\text{new})}, x^{(\text{new})}) - 2k(x^{(\text{new})}, x^{(n)}) + k(x^{(n)}, x^{(n)}).$$

And this gives us a kernelized KNN.

4.3.5. Advantages/disadvantages of the kernelized SVM. They are the same as for the linear SVM (section 4.2.5) except that now, we can tackle problems where the data is not linearly separable.

4.4. Examples in R

4.4.1. Toy example for linear SVM. We create here our own data. We fix the random seed to be able to do the same drawing many times. The matrix `x` is made of normal random variables. The matrix `y` is made of -1 and $+1$, this matrix contains the labels for the components of `x`. The components of `x` labelled with a $+1$ are moved (we add the vector $(1,1)$). Then, we plot the points with different colors according to their labels, the `pch=19` option makes nice fat dots.

```
set.seed(1011)
x = matrix(rnorm(40), 20, 2)
y = rep(c(-1, 1), 10)
x[y == 1,] = x[y == 1,] + c(1,1)
plot(x, col = y + 3, pch = 19)
```

We transform the data into a data-frame. We call `svm` on this data-frame, using `y` as the response variable (a.k.a. output variable). Here the cost parameter is 10 and we ask `svm` not to standardize the variables. The "`linear`" option is the default option when we look for a hyperplane boundary.

```
library(e1071)
dat = data.frame(x, y = as.factor(y))
svmfit = svm(y ~ ., data = dat, kernel = "linear", cost = 10, scale = FALSE)
print(svmfit)
plot(svmfit, dat)
```

We can see the summary with `svmfit`. Here the number of support vectors is 6 (it accounts for the points closest to the boundary and the points on the wrong side of the boundary).

We want a nicer plot so we create our own grid using a function called `make.grid`. It takes in your data matrix x, as well as an argument n which is the number of points in each direction. Here you're going to ask for a 75×75 grid. Within this function, you use the apply function to get the range of each of the variables in x. Then for both x1 and x2, you use the `seq` function to go from the lowest value to the upper value to make a grid of length n. As of now, you have x1 and x2, each with length 75 uniformly-spaced values on each of the coordinates. Finally, you use the function expand.grid, which takes x1 and x2 and makes the lattice.

```
make.grid = function(x, n = 75)
{
        grange = apply(x, 2, range)
        x1 = seq(from = grange[1,1], to = grange[2,1], length = n)
        x2 = seq(from = grange[1,2], to = grange[2,2], length = n)
        expand.grid(X1 = x1, X2 = x2)
```

```
}
xgrid=make.grid(x)
```

Having made the lattice, you're going to make a prediction at each point in the lattice. With the new data xgrid, you use predict and call the response `ygrid`. You then plot and color code the points according to the classification so that the decision boundary is clear. Let's also put the original points on this plot using the points function.

The function `svmfit` has a component called index that tells which are the support points. You include them in the plot by using the `points` function again. Caution: the support points will appear in a box, as well as the points on the wrong side of the boundary, and as well as other points (`R` malfunction?).

```
ygrid = predict(svmfit, xgrid)
plot(xgrid, col = c("red","blue")[as.numeric(ygrid)], pch = 20, cex = .2)
points(x, col = y + 3, pch = 19)
points(x[svmfit$index,], pch = 5, cex = 2)
```

We now want to extract the coefficient of our hyperplane/boundary from the variable `svmfit`. We need to explain the maths behind `R` a little bit more. Now that we have the support points (whose indexes are `svmfit$index`), we could compute what is the best hyperplane between them ... we would end up with the same hyperplane we computed before. Suppose, for the sake of simplification, the indexes of the support points are $1,2,\dots,P$. Then, when a new point x_{new} comes by, the decision rule is

$$t_{\text{new}} = \text{sign}(\sum_{n=1}^{P} \alpha_n t^{(n)} (x^{(n)})^T x_{\text{new}} + \beta_0),$$

for some coefficients α_n, β_0 (remember that $t^{(n)} \in \{+1,-1\}$). Now, `svmfit$coefs` gives us the coefficients $(\alpha_1 t^{(1)},\dots,\alpha_P t^{(P)})$ and `svmfit$rho` gives is the coefficient β_0. If, we run the commands

```
beta = t(svmfit$coefs)%*%x[svmfit$index,]
beta0 = svmfit$rho
```

we get the value of $-\beta_0$ `in beta0` and the value of $\sum_{n=1}^{P} \alpha_n t^{(n)} (x^{(n)})^T$ in `beta`. The `abline` command draw a line in the current plot (its argument are the intercept and the slope). Let us write $\sum_{n=1}^{P} \alpha_n t^{(n)} (x^{(n)})^T = [\beta_1, \beta_2]$. Our boundary has the equation: $\beta_0 + \beta_1 x_1 + \beta_2 x_2 = 0$. So it is a line with intercept $-\beta_0/\beta_2$ and slope $-\beta_1/\beta_1$. The following commands draw the boundary and lines parallel to the boundary, spot on the support vectors

```
abline(beta0 / beta[2], -beta[1] / beta[2])
abline((beta0 - 1) / beta[2], -beta[1] / beta[2], lty = 2)
abline((beta0 + 1) / beta[2], -beta[1] / beta[2], lty = 2
```

Here, we use the fact that the support vectors $x^{(i)}$ are such that $[\beta_1, \beta_2] \times x^{(i)} + \beta_0 = \pm 1$.

4.4.2. Exercise on a toy example (using the commands we have learned above).

(1) Draw 40 points with uniform law in the square $[0,1] \times [0,1]$. Shift 20 of them by the vector $(0, 0.8)$ (label them with 1 and label the rest of the points with -1). Using `svm`, run a linear SVM classification on these points (with the `cost` parameter set to 10). Make a drawing of the result.
(2) We now want a better drawing. Create a grid covering a rectangle such that our points are at a distance bigger than 0.5 of the edges. Color the points in the grid according to their classification (by our classifier, built in the previous question). Draw the original points. Draw a box around the support points
(3) Draw the boundary and lines parallel to the boundary, going through the support points.
(4) Try again the previous question with a cost parameter fixed to 100. The abnormal points should disappear.

4.4.3. Toy example for kernelized SVM. We load a `R` file and look at its components.

```
load(file = "ESL.mixture.rda")
names(ESL.mixture)
ESL.mixture$x
ESL.mixture$y
```

In `x`, we have points (in $\mathbb{R}^2$) and in `y`, we have labels (0 or 1). We plot the points, coloring them according to their labels.

```
plot(x, col = y + 1)
```

We then transform the data into a data-frame.

```
dat = data.frame(y = factor(y), x)
```

We fit a kernelized SVM on the data.

```
fit = svm(factor(y) ~ ., data = dat, scale = FALSE, kernel = "radial", cost = 5)
```

Here the kernel chosen is the Gaussian one (see Section 4.3.2), also called Radial kernel.

It's time to create a grid and make your predictions. These data actually came supplied with grid points. If you look down on the summary on the names that were on the list, there are 2 variables px1 and px2, which are the grid of values for each of those variables (we will go back to this command, with an example, in the next Section). You can use expand.grid to create the grid of values. Then you predict the classification at each of the values on the grid.

```
xgrid = expand.grid(X1 = px1, X2 = px2)
ygrid = predict(fit, xgrid)
```

Finally, you plot the points and color them according to the decision boundary. You can see that the decision boundary is non-linear. You can put the data points in the plot as well to see where they lie.

```
plot(xgrid, col = as.numeric(ygrid), pch = 20, cex = .2)
points(x, col = y + 1, pch = 19)
```

Now, we want to plot the decision boundary. First, we predict our fit on the grid by using the command

```
func = predict(fit, xgrid, decision.values = TRUE)
```

where we use the option `decision.values=TRUE` because you want to get the actual function, not just the classification (you can see that `func` is made of zeros and ones). Next, we can transform these zeros and ones into positive or negative values (reflecting the algebraic distance to the boundary). Is now time to use the `contour` function. We reshape the data in `func` into a matrix of the right proportions (69×99 because `length(px1)` is 69 and `length(px2)` is 99). We want a line separating the positive and negative values of `func` so we choose `level=0`. The option `add=TRUE` means we add the contour to the current plot.

```
contour(px1, px2, matrix(func, 69, 99), level = 0, add = TRUE)
```

We retry the experiment with different paramaters `cost` (see Figure 4.4.1). When the cost is high, the algorithm does not like to have points of the training set which are not rightly classified, so the boundary gets very wiggly. It is clear that a very wiggly boundary will not get you a good accuracy on a test set (we say that the generalization error is likelily to be high).

4.4.4. Car acquisition. We are interested in the file[1] `social.csv`. We download it and keep only the columns from 3 to 5.

```
dataset = read.csv('social.csv')
dataset = dataset[3:5]
```

[1] `https://www.kaggle.com/rakeshrau/social-network-ads?select=Social_Network_Ads.csv`

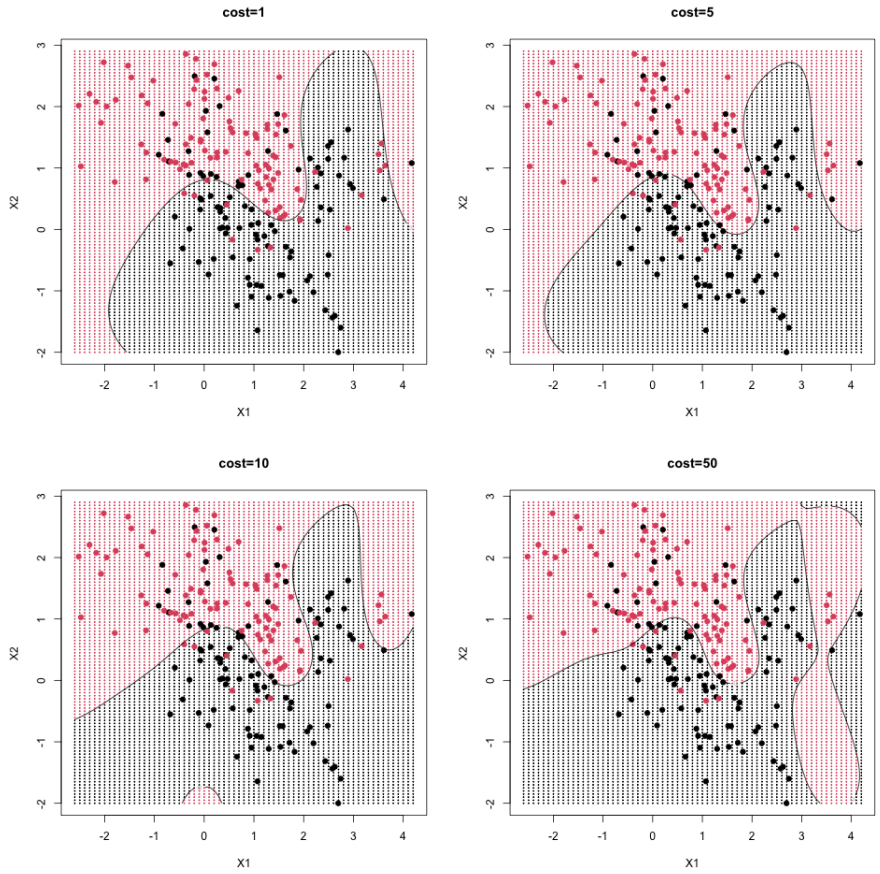


FIGURE 4.4.1. Various costs.

For various individuals, we have their age, salary (data dating from one year before the present) and whether they have bought a car in the last year (1 if have bought a car, 0 if not). The goal is to use this data to make predictions about car acquisition for individuals. Imagine you have this kind of data for various web users. You want to predict if these people are likely to buy a new car (say, in the next year). If you have such a prediction, you can perform target advertising on them (namely, show them offers for buying new cars).

We can encode the target feature as a factor and then split the data into two sets.

```
dataset$Purchased = factor(dataset$Purchased, levels = c(0, 1))
 library(caTools)
split = sample.split(dataset$Purchased, SplitRatio = 0.75)
training_set = subset(dataset, split == TRUE)
test_set = subset(dataset, split == FALSE)
```

We then want to re-scale the data. In the case of the SVM, re-scaling the data has the effect of re-scaling the decision boundary, it does not affect the decision function (i.e. how we would classify a new incoming data point). We do this re-scaling merely to have a nice drawing in the end.

```
training_set[-3] = scale(training_set[-3])
test_set[-3] = scale(test_set[-3])
```

We fit the SVM to the training set.

```
classifier = svm(formula = Purchased ~ .,
        data = training_set,
```

```
        type = 'C-classification',
        kernel = 'linear')
# Predicting the Test set results
y_pred = predict(classifier, newdata = test_set[-3])
# Making the Confusion Matrix
cm = table(test_set[, 3], y_pred)
```

The option `kernel='linear'` means we are looking for a decision boundary which is a hyperplane (we will see later that it could be something else). We then want to plot the data points and the decision boundary. We put the coordinates we want to draw into `X1` and `X2`. The command `expand.grid` creates a grid with the available coordinates. For example `expand.grid(c(1,2),c(1,2,3))` will return the following data-frame

...	...
1	1
1	2
1	3
2	1
2	2
2	3

```
set = training_set
X1 = seq(min(set[, 1]) - 1, max(set[, 1]) + 1, by = 0.01)
X2 = seq(min(set[, 2]) - 1, max(set[, 2]) + 1, by = 0.01)
grid_set = expand.grid(X1, X2)
colnames(grid_set) = c('Age', 'EstimatedSalary')
y_grid = predict(classifier, newdata = grid_set)
```

The last line above makes a predictions for each point in our grid. We then plot the data points.

```
 plot(set[, -3],
        main = 'SVM (Training set)',
        xlab = 'Age', ylab = 'Estimated Salary',
        xlim = range(X1), ylim = range(X2))
contour(X1, X2, matrix(as.numeric(y_grid), length(X1), length(X2)), add = TRUE)
```

The command `contour` creates a contour plot for the classification values. As those are 0 or 1, this draws the decision boundary. We then color the data points accordingly to their labels and color the grid points accordingly to the decision function.

```
points(grid_set, pch = '.', col = ifelse(y_grid == 1, 'coral1', 'aquamarine'))
points(set, pch = 21, bg = ifelse(set[, 3] == 1,'coral1', 'aquamarine'))
```

The color red corresponds to label 1 (car purchased) and the color blue corresponds to label 0 (car not purchased).

4.4.5. Digit recognition (activity). The MNIST database[2] of handwritten digits, available from this page, has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image. We want to try various algorithms on the handwriting recognition problem. Here we want our algorithm to recognize digits.

The original dataset is in a format that is difficult for beginners to use (it is made of pictures in grey scale). The `mnist_train.csv` file contains the 60,000 training examples and labels. The `mnist_test.csv` contains 10,000 test examples and labels. Each row consists of 785 values: the first value is the label (a number from 0 to 9) and the remaining 784 values are the pixel values (a number from 0 to 255).

We can visualize the first 36 digits in the training set by using the following code.

[2] `https://www.kaggle.com/oddrationale/mnist-in-csv`

```
train<-read.csv('mnist_train.csv')
test<-read.csv('mnist_test.csv')
x_train<-train[-1]
y_train<-train[,1]
par(mfcol=c(6,6))
par(mar=c(0, 0, 3, 0), xaxs='i', yaxs='i')

for (idx in 1:36)
{
        im <- x_train[idx,]
        im<-as.numeric(im)
        im1<-matrix(im,28,28,byrow=F)
        im2<-im1
        for (j in 1:28)
        {
                im2[,j]=im1[,28-j+1]
        }
        image(1:28, 1:28, im2, col=gray((0:255)/255),xaxt="n", main=paste(y_train[idx]))
}
```

And we get the picture in Figure 4.4.2. Try a linear SVM classifier on this set. Hint: the data set is huge,

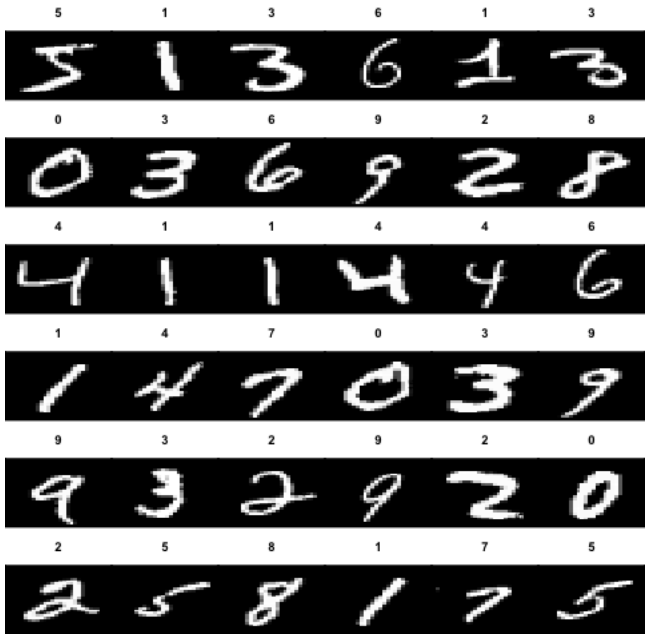


FIGURE 4.4.2. MNIST dataset

so start by taking into account only a fraction of the training set.

4.4.6. Summary of useful R commands. SVM classification:

```
library(e1071)
svmfit = svm(y ~., data = dat, kernel = "linear", cost = 10, scale=FALSE)
```

where the data is into `dat` with labels in `dat$y`. The kernel can be linear (`linear`) , Gaussian (`radial`), polynomial (`polynomial`), ... The parameter `cost` determines how much we pay for a point of the

training set on the wrong side of the boundary. The `scale` option allows to re-scale the data if we want (choose `FALSE` if you do not want to re-scale).

Create a grid with x-coordinates in `px` and y-coordinates in `py`:

```
expand.grid(X1=px,X2=py)
```

Plot a line with slope `a` and intercept `b`:

```
abline(b,a,lty=2)
```

(`lty` is a graphical parameter).

Draw a contour line of a function f (x-coordinates in `px1`, y-coordinates in `px2`, `M` is a matrix with the values of f on the grid `expand.grid(X1=px,X2=py)`, `level`=level of the contour, `add=TRUE` if we want to add the contour to an existing plot):

```
contour(px1, px2, matrix(func, 69, 99), level = 0, add = TRUE)
```

Convert a $n \times p$ matrix `im2` in to an image (gray-scale)

```
image(1:n, 1:p, im2, col=gray((0:255)/255),xaxt="n")
```

(`xaxt` is a graphical parameter).

CHAPTER 5

Bayes classifier and naive Bayes classifier.

5.1. The Bayes classifier

5.1.1. Introduction. Suppose you are the webmaster of a website hosting newsgroups (a total of C newsgroups). This website is very popular and the flow of posts per minute is huge. You want to attribute a label to each new post (wether it belongs to newsgroup number n). Our aim here is to compute the predictive probabilities for each of the C potential classes. These probabilities can then form the basis of a decision-making process (we decide that a post belongs to some class and check whether the author of the post classified it correctly). The typical quantities are $C = 20$ and the number of posts to classify each day is 20000. So, there is no way the classification could be made by a human.

We need to encode each post as a vector of numerical values. The most common way to do this is to use a bag-of-words model. If the total number of unique words in all documents is M (i.e. the vocabulary used consists of M words), each document is represented as a M-dimensional vector. The vector for the n-th post $x^{(n)}$ is made up of the counts of the number of times each word appears in the post. The feature (or coordinate) $x_m^{(n)}$ is thus the number of time the word number m appears in the post number n.

The general model is the following. We have a data set X. Each component of X is a post x and has a label y (which is correct). The set of labels is called Y. We summarize the data by writing $\mathscr{D} = (X, Y)$. Now, we see a new post coming in. We will treat this new incoming as a random variables. We call it $X^{(\text{new})}$. The correct label of the new incoming is treated as a random variable. We call it $Y^{(\text{new})}$.We would like to compute

$$\mathbb{P}(Y^{(\text{new})} = c | X^{(\text{new})} = x^{(\text{new})}, \mathscr{D}).$$

We apply Bayes' Theorem to the probability above, without thinking twice about what it could mean and we get

$$\mathbb{P}(Y^{(\text{new})} = c | X^{(\text{new})} = x^{(\text{new})}, \mathscr{D}) = \frac{\mathbb{P}(X^{(\text{new})} = x^{(\text{new})} | Y^{(\text{new})} = c, \mathscr{D}) \mathbb{P}(Y^{(\text{new})} = c | \mathscr{D})}{\mathbb{P}(X^{(\text{new})} = x^{(\text{new})} | \mathscr{D})}.$$

We then use the law of total probability to expand the denominator $\mathbb{P}(X^{(\text{new})} = x^{(\text{new})} | \mathscr{D})$ and we obtain

$$\mathbb{P}(Y^{(\text{new})} = c | X^{(\text{new})} = x^{(\text{new})}, \mathscr{D}) = \frac{\mathbb{P}(X^{(\text{new})} = x^{(\text{new})} | Y^{(\text{new})} = c, \mathscr{D}) \mathbb{P}(Y^{(\text{new})} = c | \mathscr{D})}{\sum_{c'=1}^{C} \mathbb{P}(X^{(\text{new})} = x^{(\text{new})} | Y^{(\text{new})} = c', \mathscr{D}) \mathbb{P}(Y^{(\text{new})} = c' | \mathscr{D})}. \tag{5.1}$$

If we define values for the probabilities of the kind

$$\mathbb{P}(X^{(\text{new})} = x^{(\text{new})} | Y^{(\text{new})} = c, \mathscr{D}), \mathbb{P}(Y^{(\text{new})} = c | \mathscr{D})$$

(for all c in C) then we can compute the probability we are interested in. Our task is now to find *credible* values for the probabilities above.

5.1.2. Likelihood (probability of the kind $\mathbb{P}(X^{(\textbf{new})} = x^{(\textbf{new})} | Y^{(\textbf{new})} = c, \mathscr{D})$) (or class-conditional distribution). The likelihood term $\mathbb{P}(X^{(\text{new})} = x^{(\text{new})} | Y^{(\text{new})} = c, \mathscr{D})$ is a distribution specific to the c-th class, evaluated at $x^{(\text{new})}$. To create our Bayes classifier, we need to define C of these class-conditional distributions. We could use the same type of distribution for each class (or not).

Once we have chosen the distribution for the c-class, we need to choose its parameters. For example, if we have chosen a Gaussian distribution, we need to choose the mean and covariance matrix. This will be done using the data (for example, using a MLE).

5.1.3. Prior class distribution ($\mathbb{P}(Y^{(\mathbf{new})} = c|\mathscr{D})$). This probability is the probability that an object belongs to class c, conditionally on the data $\mathscr{D}$, and without seeing the object itself. The choice of this probability enables us to specify any prior beliefs we have in the class c. There are technical restrictions of course:

$$\begin{cases} \mathbb{P}(Y^{(\text{new})} = c|\mathscr{D}) > 0\,, \forall\, c\,, \\ \sum_{c'=1}^{C} \mathbb{P}(Y^{(\text{new})} = c'|\mathscr{D}) = 1\,. \end{cases}$$

Two popular choices are the following

$$\begin{cases} \text{uniform prior} & : \quad \mathbb{P}(Y^{(\text{new})} = c|\mathscr{D}) = \frac{1}{C}\,, \\ \text{class size prior} & : \quad \mathbb{P}(Y^{(\text{new})} = c|\mathscr{D}) = \frac{N_c}{N}\,, \end{cases} \tag{5.2}$$

where $N = \#X$ (the size of the data) and $N_c = \#\{y \in Y : y = c\}$ (number of points in the data with label c).

5.2. Naive Bayes classifier (NBC).

5.2.1. Introduction. In this Section, we discuss how to classify vectors of discrete-valued features. Each vector x belongs to $\{1,2,\dots,K\}^D$ (K is the number of possibilities for each feature). Remember we have to specify the class-conditional distribution $\mathbb{P}(X^{(\text{new})} = x^{(\text{new})}|Y^{(\text{new})} = c, \mathscr{D})$. The simplest approach is to assume the features are conditionally independent given the class label. This allows us to write

$$\mathbb{P}(X^{(\text{new})} = x^{(\text{new})}|Y^{(\text{new})} = c, \mathscr{D}) = \prod_{j=1}^{D} \mathbb{P}(X_j^{(\text{new})} = x_j^{(\text{new})}|Y^{(\text{new})} = c, \mathscr{D})\,. \tag{5.1}$$

The resulting model is called naive Bayes classifier (NBC). This model is called "naive" because we do not expect the features to be independent. Think about the newsgroup example we discussed before. We could have classes like: "foreign policy", "economy", "culture", ... If we have a post belonging to the culture class, the number of times we see the word "novel" in this post is clearly dependent of the number of times we see the word "writer" in this post. However, this model often works well in practice. One reason is that the model is quite simple. Its number of parameters is of order $C \times K$ (this is the number of probabilities you have to specify in the right-hand side of Equation (5.1) and hence it is relatively immune to over-fitting.

EXAMPLE 5.1. In the case of real-valued features, we can use a Gaussian distribution:

$$\mathscr{L}(X^{(\text{new})}|Y^{(\text{new})} = c, \mathscr{D}) \text{ has density } x^{(\text{new})} \mapsto \prod_{j=1}^{D} \mathscr{N}(x_j^{(\text{new})}|\mu_{j,c}, \sigma_{j,c}^2)\,.$$

⚠ We write $\mathscr{L}$ for law. The notation is misleading: $t \mapsto \mathscr{N}(t|\mu,\sigma^2)$ is the density of the Gaussian $\mathscr{N}(\mu,\sigma^2)$. Remember that $x^{(\text{new})} = (x_1^{(\text{new})},\dots,x_D^{(\text{new})})^T$. The Gaussians $\mathscr{N}(x_j^{(\text{new})}|\dots)$ depend on c so we write $\mu_{j,c}$ and $\sigma_{j,c}^2$ for their mean and variance respectively.

EXAMPLE 5.2. In the case of binary features ($x_j \in \{0,1\}$, $\forall\, j$), we can use a Bernoulli distribution (as it is binary, it cannot be anything else):

$$\mathscr{L}(X^{(\text{new})}|Y^{(\text{new})} = c, \mathscr{D}) \text{ has density } x^{(\text{new})} \mapsto \prod_{j=1}^{D} \mathscr{B}(x_j^{(\text{new})}|\mu_{j,c})\,.$$

This is sometimes called a multivariate Bernoulli naive Bayes model.

⚠ Here, $t \mapsto \mathscr{B}(t|p)$ is the density of the Bernoulli law of parameter p. This density is taken with respect to the measure $\delta_0 + \delta_1$.

EXAMPLE 5.3. In the case of categorical features ($x_j \in \{1,2,\dots,K\}$), we can use the "multinoulli" distribution:

$$\mathscr{L}(X^{(\text{new})} = x^{(\text{new})}|Y^{(\text{new})} = c, \mathscr{D}) = \otimes_{1\le j\le D} \text{Cat}(X_j^{(\text{new})}|\mu_{j,c})\,,$$

where $\text{Cat}(X_j^{(\text{new})}|\mu_{j,c})$ is the law such that $\mathbb{P}(X_j^{(\text{new})} = k) = \mu_{j,c}(k)$ (with $\sum_{k=1}^{K} \mu_{j,c}(k) = 1$). The product $\otimes$ above means the component of $X^{(\text{new})}$ are independent and each component has a law $\text{Cat}(\dots|\mu_{j,c})$.

5.2.2. Maximum Likelihood Estimator for the Naive Bayes Classifier. Whatever model we have chosen in Section 5.1.2 and Section 5.1.3, we can always write it in the following form. The set $\mathscr{D}$ is made of i.i.d. points (X,Y) having the following law:

$$\begin{cases} \mathscr{L}(X|Y=c,\mathscr{D}) \text{ is a distribution parametrized by } \theta_c\,, \\ \mathbb{P}(Y=c|\mathscr{D}) = \pi_c \text{ (with } \pi_1,\dots,\pi_C \geq 0 \text{ such that } \pi_1+\cdots+\pi_C = 1). \end{cases}$$

We then have that the density of $(X^{(\text{new})}, Y^{(\text{new})})$ taken in (x,y) is equal to

$$\mathbb{P}(x,y|(\theta_c)_{1\leq c\leq C},(\pi_c)_{1\leq c\leq C}) = \prod_{c=1}^{C} \pi_c^{\mathbb{1}(y=c)} \prod_{j=1}^{D}\prod_{c=1}^{C} \mathbb{P}(x_j|\theta_{j,c})^{\mathbb{1}(y=c)}$$

(this density is taken with respect to $\delta_1+\cdots+\delta_C$ concerning the y-part). Hence the log-likelihood is (we suppose we can write $\mathscr{D} = (x^{(i)}, y^{(i)})_{1\leq i\leq N}$)

$$\log\mathbb{P}(\mathscr{D}|\theta) = \sum_{c=1}^{C} N_c \log(\pi_c) + \sum_{j=1}^{D}\sum_{c=1}^{C} \sum_{i:y^{(i)}=c} \log\mathbb{P}(x_j^{(i)}|\theta_{j,c})\,,$$

where (as in Equation (5.2)) N_c is the number of points with label c in the data. We have a sum of terms concerning π plus a sum of terms concerning θ. Hence, we can optimize these parameters separately.

The MLE for the class prior is

$$\widehat{\pi}_c = \frac{N_c}{N}\,,\ \forall c\,.$$

The MLE for the likelihood depends on the type of distribution we use for each feature. Suppose we have decided that the law of x_j knowing $y=c$ is a Bernoulli with parameter $\theta_{j,c}$, then the MLE is

$$\widehat{\theta}_{j,c} = \frac{N_{j,c}}{N_c}$$

where $N_{j,c} = \#\{i : y^{(i)} = 1, x_j^{(i)} = 1\}$ (among the data points such that $y^{(i)}=1$, we count how many are such that $x_j^{(i)} = 1$).

5.2.3. Newsgroup example (continued). We use multinomials for the class-conditional distribution:

$$\mathbb{P}(X^{(\text{new})} = x^{(\text{new})}|Y^{(\text{new})} = c,\mathscr{D}) = \left(\frac{s_n!}{\prod_{m=1}^{M} x_m^{(n)}!}\right) \prod_{m=1}^{M} q_{c,m}^{x_m^{(n)}}\,, \tag{5.2}$$

where $s_n = \sum_{m=1}^{M} x_m^{(n)}$ and $q_c = (q_{c,1},\dots,q_{c,M})^T$ with $q_m \geq 0$ $(\forall m)$ and $q_{c,1}+q_{c,2}+\cdots+q_{c,M} = 1$. Remember that the vocabulary is made of M word, for the post number n, $x_m^{(n)}$ is the number of times the word number m appears in the post. In this case, the MLE is (for class c)

$$q_{c,m} = \frac{\sum_{n:y^{(n)}=c} x_m^{(n)}}{\sum_{m'=1}^{M}\sum_{n:y^{(n)}=c} x_{m'}^{(n)}}\,. \tag{5.3}$$

EXERCISE 5.1. Suppose the vocabulary has 20000 words. We compute the complexity of our model. Suppose we find the complexity is roughly proportional to $\sqrt{200000}$.We will then assume that the time needed to compute a prediction for a new post is $\tau_0 \times \sqrt{20000}$ with $\tau_0 = 1.10^{-6}s$. How much time do we spare when computing the prediction for one post if we use the "naive" assumption? A) 1s B) 7min C) 20s D) more than one month (Answer in the remark)

REMARK 5.4. The "naive" assumption simplifies the problem greatly. The number of parameters required to define each class-conditional is roughly equal to the number of words (M). If we looked at pair-wise dependencies, we would need around M^2 parameters. Given that a typical vocabulary might incude 50000 words, this is already a significant challenge.

In the exercise above, using the "naive" assumption, we have a model of complexity 20000 instead of 20000×20000. So the time spared is (in seconds): $20000^2\times 0.000001 - 20000\times 0.000001 = 399.8\text{s} \approx 7\text{mn}$. The time used by one prediction (under the "naive" assumption) is: $20000\times 0.000001 = 0.02\text{s}$

5.2.4. Bayesian NBC. Let us carry on with the above example. Suppose a particular word (with number m_0) never appear in documents from a particular class (with number c_0). For example, we can safely assume the word "weasel" does not appear in the "foreign policy" section, though it might very well appear in the "science and nature" section. In particular, imagine that Donald Trump tweets "Kim Jong-un is a weasel". Immediately, there will be posts commenting this tweet. These have to be classified in the "foreign policy" section.

EXERCISE 5.2. Where are this posts going to be classified ? A) foreign policy B) science and nature C) anywhere

If we look at q_{c_0,m_0} in the formula above (Equation (5.3)), we see that, at the numerator, we have a sum of zeros. So

$$q_{c_0,m_0} = 0\,.$$

Which implies

$$\mathbb{P}(X^{(\text{new})} = x^{(\text{new})} | Y^{(\text{new})} = c, \mathscr{D}) = 0$$

(from Equation (5.2) above). Now, for a new post coming in, we compute (using our "master" Equation (5.1))

$$\begin{aligned} \mathbb{P}(Y^{(\text{new})} = c_0 | x^{(\text{new})}, \mathscr{D}) &= \mathbb{P}(X^{(\text{new})} = x^{(\text{new})} | Y^{(\text{new})} = c, \mathscr{D}) \times \dots \\ &= 0\,. \end{aligned}$$

Alas, our model cannot possibly classify these new posts in the "foreign policy" section. This is called the zero probability problem. So, there is something wrong in our model and we need to re-think the whole thing.

One possible solution. The thing is, we can overcome this problem by using a Bayesian procedure. We place a prior density on q that encodes the belief that all the probabilities are positive. Once we have defined this prior, we can set q with the MAP estimate (Maximum A Posteriori). Let us have a look at the details. For the prior, we choose the Dirichlet density, defined as

$$\mathbb{P}(q_c | \alpha) = \frac{\Gamma(\sum_{m=1}^M \alpha_m)}{\prod_{m=1}^M \Gamma(\alpha_m)} \prod_{m=1}^M q_{c,m}^{\alpha_m - 1}\,, \forall c\,,$$

where Γ is the gamma function ($\Gamma(x) = \int_0^{+\infty} t^{x-1} e^{-t}\,dt$, $\Gamma(n) = n!$) and $\alpha = (\alpha_1, \dots, \alpha_M)$ is a parameter to be chosen. We will simplify further by assuming that $\alpha_m = \alpha_0$ ($\forall m$). After some computation, we get the following MAP estimate

$$q_{cm} = \frac{\alpha_0 - 1 + \sum_{n:y^{(n)}=c} x_m^{(n)}}{M \times (\alpha_0 - 1) + \sum_{m'=1}^M \sum_{n:y^{(n)}=c} x_{m'}^{(n)}}\,. \tag{5.4}$$

This technique is often referred to as smoothing or Laplace smoothing.

5.3. Advantages/disadvantages of NBC

Advantages.

- Easy implementation.
- It is fast and it is efficient when used on large datasets
- The algorithm is not affected by noise (outliers are not a problem either). It is also immune to over-fitting.
- It is easy to take into account new data.

Disadvantages.

- Sometimes it is too simple. Features can be really not independent and this will impact your result.

5.4. Examples in R

5.4.1. The `naiveBayes` command. We are going to use the `naiveBayes` command. It comes with the library `e1071`. It has the following form

```
naiveBayes(formula, data, laplace = alpha, subset, na.action = na.pass)
```

- The `formula` is of the kind `Y~` (see examples below).
- The `data` is a data-frame of numeric or factor variables.
- The option `laplace` provides a smoothing effect.
- Using the `subset` option, you can use a selected subset of your data (based on some boolean filter). Or you can split your data beforehand.
- The command `na.action` is used to decide what to do in case of missing values (here, we choose pass).

The variables in the data-frame different from the one we want to predict (`Y`) can be integers, factors, real numbers. Suppose they are D-dimensional vectors.

If they are integers or factors, `R` will assume they have a limited of possible values and choose the following class-conditional distribution:

$$\mathbb{P}(X^{(\text{new})} = x^{(\text{new})} | Y^{(\text{new})} = c, \mathscr{D}) = \prod_{j=1}^{D} \frac{\sum_{n:y^{(n)}=c} \mathbb{1}(x_j^{(n)} = x_j^{(\text{new})})}{\#\{n : y^{(n)} = c\}}.$$

For each j, we have the number of times we have the feature $x_j^{(\text{new})}$ in the points of the dataset such that $y^{(n)} = c$, divided by the numbers of points of the dataset in class c. If $x_j^{(\text{new})}$ is never seen at the j-th coordinate in the dataset, we run again into a zero probability problem. This can be avoided by adding a parameter α (the one in `laplace=...`). We then get

$$\mathbb{P}(X^{(\text{new})} = x^{(\text{new})} | Y^{(\text{new})} = c, \mathscr{D}) = \prod_{j=1}^{D} \frac{\alpha + \sum_{n:y^{(n)}=c} \mathbb{1}(x_j^{(n)} = x_j^{(\text{new})})}{M_j \alpha + \#\{n : y^{(n)} = c\}}$$

where M_j is the number of values that can be taken by $x_j^{(n)}$.

If the variables in the data-frame are real numbers, `R` will assume they have a continuous distribution and that this distribution is Gaussian, Furthermore, `R` will estimate the mean and variance of each Gaussian via Maximum-Likelihood.

More details on this command on `https://cran.r-project.org/web/packages/naivebayes/vignettes/intro_naivebayes.pdf`.

5.4.2. Iris dataset. We once again use the iris dataset.

```
data(iris)
str(iris)
```

We need some libraries.

```
library(e1071)
library(caTools)
library(caret)
```

We split the data into a train set and a test set

```
split <- sample.split(iris, SplitRatio = 0.7)
train_cl <- subset(iris, split == "TRUE")
test_cl <- subset(iris, split == "FALSE")
```

We train the data (without specifying any fancy option, we will see this later).

```
set.seed(120)
classifier_cl <- naiveBayes(Species ~ ., data = train_cl)
classifier_cl
```

The computer informs us of what a priori probability it has chosen (for $\mathbb{P}(Y^{(\text{new})} = c|\mathscr{D})$ and what estimates it has made for the means and variances of the class-conditional distributions (assumed to be Gaussian). We can then make our prediction on the test set and check what are the performances.

```
y_pred <- predict(classifier_cl, newdata = test_cl)
cm <- table(test_cl$Species, y_pred)
cm
confusionMatrix(cm)
```

5.4.3. Naïve Bayes on SMS data. We download the data[1].

```
sms_raw <- read.csv("sms_spam.csv", stringsAsFactors = FALSE)
str(sms_raw)
```

This shows us that we have two features - one containing the categorical data of either spam or ham called type and one called text, which has the actual text messages contained within. As the data in type is currently not represented as categorical data, we can set that manually:

```
sms_raw$type <- factor(sms_raw$type)
str(sms_raw$type)
```

We then have to clean and standardize the data. The `Vcorpus` function creates a corpus, or a "body" of text documents as a list object.

```
library(tm)
sms_corpus <- VCorpus(VectorSource(sms_raw$text))
typeof(sms_corpus)  #Just to show that it is a list
```

We then want to perform standardization actions:

- Replace capital letters by lowercase letters (performed by the `tm_map(content_transformer(tolower))` function).
- Remove numbers (performed by the `tm_map(RemoveNumbers)` function).
- Remove stop words (such as: to, but, for, ...) (performed by the `tm_map(removeWords(stopwords(),...))` function).
- Remove punctuation (performed by the `tm_map(removePunctuation)` function).
- Stem words. This refers to reducing words with the same root to the root itself. For example: 'jump', 'jumping', 'jumped' ... are reduced to 'jump' (performed by the `tm_map(stemDocument)` function).
- Get rid of the white spaces (performed by the `tm_map(stripWhitespace)` function).

To simplify the code, we will use the pipe operator (%>%). This enables us to write, for any function `f` and any argument `x`: '`x %>% f`' in place of '`f(x)`'[2]. This is particularly useful when we have many functions `f`, `g`, `h`. We can replace '`f(g(h(x)))`' (hard to read because there is a lot of operators) by '`x %>% h %>% g %>% f`'. We then simply write:

```
sms_corpus_clean <- sms_corpus %>%
        tm_map(content_transformer(tolower)) %>%
        tm_map(removeNumbers) %>%
        tm_map(removeWords, stopwords()) %>%
        tm_map(removePunctuation) %>%
        tm_map(stemDocument) %>%
        tm_map(stripWhitespace)
```

We can compare the some of the raw messages to some of the standardized message by using

[1] `https://www.kaggle.com/hdza1991/sms-spam`

[2] What the function does is to pass the left hand side of the operator to the first argument of the right hand side of the operator.

```
for(i in 1:10)
{
        print(as.character(sms_corpus[[i]]))
}
for(i in 1:10)
{
        print(as.character(sms_corpus_clean[[i]]))
}
```

(the result is not that clean).

We then want to transform our data using the bag-of-words technique. Suppose we have a total of M words in all the text messages and a total of N text messages. The `DocumentTermMatrix` will create a $N \times M$ matrix where the entry (i, j) is the number of occurrences of word number j in message number i.

```
sms_dtm <- DocumentTermMatrix(sms_corpus_clean)
```

We now split the data into a training set and a test set. Because the data was stored randomly, we can simply take the first 75% of our entries as our training set and the remainder can be taken as our test set.

```
sms_dtm_train <- sms_dtm[1:4169, ]
sms_dtm_test <- sms_dtm[4170:5559, ]
sms_train_labels <- sms_raw[1:4169, ]$type
sms_test_labels <- sms_raw[4170:5559, ]$type
```

A simple command allows us to see the "spam" messages account for 13% of the messages

```
sms_train_labels %>%   table %>%   prop.table
sms_test_labels %>%   table %>%   prop.table
```

A word cloud will allow to visualize which words are the more frequents (words appearing frequently are shown with a larger font size). Let us look at what happens if we use this world cloud representation for the "spam" messages and the "ham" messages.

```
  par(mfcol = c(1, 2))
spam <- sms_raw %>%   subset(type == "spam")
spamCloud <- wordcloud(spam$text, max.words = 40, scale = c(3, 0.5))
ham <- sms_raw %>%   subset(type == "ham")
hamCloud <- wordcloud(ham$text, max.words = 40, scale = c(3, 0.5))
```

(see the result in Figure 5.4.1, "spam" on the left and "ham" on the right). We can see the two clouds are substancially different.

We complete our data preprocessing by reducing the number of features in our test and training matrices (this will greatly reduce the execution time). We use the `findFreqTerms()` function

```
sms_dtm_freq_train <- sms_dtm_train %>%
        findFreqTerms(lowfreq=5) %>%
        sms_dtm_train[ , .]
sms_dtm_freq_test <- sms_dtm_test %>%
        findFreqTerms(lowfreq=5) %>%
        sms_dtm_test[ , .]
```

(we retain only the terms with frequency higher than 5%). Now we shall write a function that converts our sparse DT matrices from numeric to factor (`naiveBayes` needs categorical data)

```
sms_train <- sms_dtm_freq_train %>%   apply(MARGIN = 2, factor)
sms_test <- sms_dtm_freq_test %>%   apply(MARGIN = 2, factor)
```

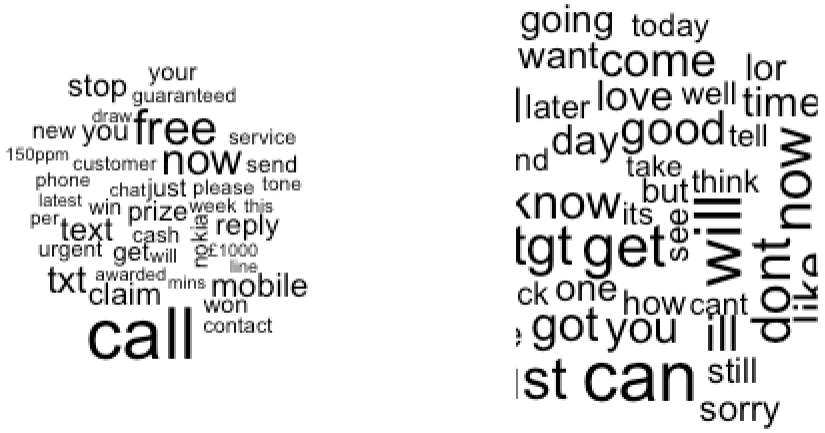


FIGURE 5.4.1. Word cloud

Now comes the fairly straightforward step of training our model on the data and then using that classifier to make predictions on the test set. This requires the `e1071` package to apply the `naiveBayes()` function:

```
library(e1071)
sms_classifier <- naiveBayes(sms_train, sms_train_labels,laplace=1)
sms_pred <- predict(sms_classifier, sms_test)
```

(by default, we set the `laplace` parameter to 1 to avoid any zero probability problem). We can test our classifier on our test set.

```
pred<-predict(sms_classifier,newdata=sms_test)
cm<-table(sms_test_labels,pred)
cm
confusionMatrix(cm)
confusionMatrix(cm)$overall[1]
```

Here, we get many statistics and we see that the accuracy is 97%.

5.4.4. Movies reviews classification (activity). Now, you can apply what you have learned on the data set[3] `IMDB_Dataset.csv`. This data set has 50000 observations of two variables ("review", "sentiment"). Each line corresponds to a movie, the "review" is a review written by an IMDB user and the "sentiment" is the overall rating of the review ("negative" or "positive"). As in the above Section, we want to use naïve Bayes to predict the sentiment, based on a review. So your task is to build a classifier that tells if a review is negative or positive.

The data set is big so most desktop computers will only be able to treat part of it (we advise to use only the first 5000 lines of the data). You can look for a possible answer in `NBC-movies.R` (provided without much comment as it is very similar to the above code). Observe that the accuracy is fairly low (84.7%). This might become better if one used more of the data.

5.4.5. Summary of useful `R` commands. The `naiveBayes` command. It comes with the library `e1071`. It has the following form

```
naiveBayes(formula, data, laplace = alpha, subset, na.action = na.pass)
```

[3] `https://www.kaggle.com/lakshmi25npathi/imdb-dataset-of-50k-movie-reviews`

- The `formula` is of the kind `Y~.`.
- The `data` is a data-frame of numeric or factor variables.
- The option `laplace` provides a smoothing effect (for `laplace=1`, we add 1 to the count of each word, in the case of document classification)
- Using the `subset` option, you can use a selected subset of your data (based on some boolean filter). Or you can split your data beforehand.
- The command `na.action` is used to decide what to do in case of missing values (here, we choose pass).

The `Vcorpus` function creates a corpus, or a "body" of text documents as a list object:

```
library (tm)
sms_corpus <- VCorpus(VectorSource(sms_raw$text))
```

(here the texts are in the column `$text` of a data-frame `sms_raw`).

Standardization actions on a text:

- Replace capital letters by lowecase letters (performed by the `tm_map(content_transformer(tolower))` function).
- Remove numbers (performed by the `tm_map(RemoveNumbers)` function).
- Remove stop words (such as: to, but, for, ...) (performed by the `tm_map(removeWords(stopwords(),...))` function).
- Remove punctuation (performed by the `tm_map(removePunctuation)` function).
- Stem words. This refers to reducing words with the same root to the root itself. For example: 'jump', 'jumping', 'jumped' ... are reduced to 'jump' (performed by the `tm_map(stemDocument)` function).
- Get rid of the white spaces (performed by the `tm_map(stripWhitespace)` function).

Pipe operator (with function `f` and input `x`): replace

```
f(x)
```

by

```
library(magrittr)  # or library(dplyr)
x %>% f
```

Create a matrix with the words counts:

```
dtm <- DocumentTermMatrix(sms)
```

(if you have a total of M words and N texts, you get a $N \times M$ matrix where the entry (i, j) is the number of occurrences of word number j in message number i).

Draw a world cloud:

```
Cloud <- wordcloud(data$text , max.words = 40, scale = c(3 , 0.5))
```

where `data$text` contains texts (`scale` is a graphical parameter).

Find frequent term in a text (here, those appearing at least five times):

```
findFreqTerms(lowfreq=5)
```

Confusion matrix (`library(caret)`): `confusionMatrix(table)` (has to be applied to a table). This confusion matrix contains a lot of information.

We can extract the accuracy with

```
confusionMatrix(table)$overall[1]
```

Create a table

```
table.liste <- table(liste)
```

If `liste=[1,2,1,1,2,2,2,1,2]` then `table.list` returns

1	2
4	5

(the counts of 1 and 2)

Compute proportions in a table:

```
prop.table(table.liste)
```

returns

1	2
0.4444...	0.5555...

CHAPTER 6

Neural networks

6.1. Introduction

6.1.1. Foreword. Neural Network is a network of idealized neurons that has the ability to learn by example. The information processing inside the network is inspired by the biological neuron system. It is formed by a large number of connected elements, known as neurons. These neurons process the information in a parallel way and in a non-linear fashion. A neural network is said to be adaptive because we can change the links between the neurons (these links can be non-existent, have certain weights, and so on).

The neural networks were designed to solve problems which are easy for humans and difficult for machines such as classifying pictures (as the cat/dog picture at the beginning of this course. This kind of problem is referred to as pattern recognition. Its application ranges from optical character recognition to object detection.

There is mathematical results known as universal approximation theorem that state, essentially, that any function of the inputs can be approximated by a neural network, if the number of neuron goes to infinity (see [**Cyb89**], [**Hor91**], [**Hay98**], [**Has95**], [**PYLS20**]).

In this chapter, we will cover the following topics:

- Structure of the network.
- Fitting the neural network (a.k.a. learning from examples): forward propagation, backward propagation.
- Implementation of the neural network in R.

6.1.2. Biological neurons. Warren McCulloch and Walter Pitts developed a mathematical model of a neuron (see the classical paper [**MP44**]). In this model, a neuron takes electrical inputs through its

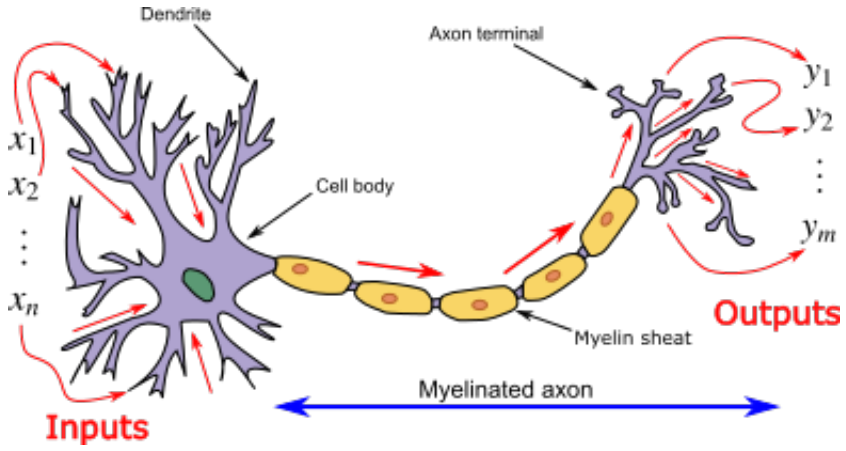


FIGURE 6.1.1. Biological neuron (source: Wikipedia)

dendrites and produces electrical outputs through its axons (see Figure 6.1.1). This model is known as the McCulloch-Pitts neural model.

One particular detail of the information processing by the neuron is that the neuron sums the inputs, it will transmit something through the dendrites if the sum reaches a certain threshold. This justifies the use of the sigmoid function later on (Section 6.1.3 and Remark 6.3).

6.1.3. Model neurons in a neural network. Neural network algorithms are inspired by this model and are designed to perform a particular task or function. Before being able to perform its task, a network must learn through examples. As the network is a set of connected input/outputs unit in which each connection as a weight associated with it, the learning phase will consist of adjusting the weights to predict the correct class label of the given inputs.

6.2. Structure of the network

For the sake of simplicity, we will study networks with only one hidden layer but R allow you to choose the number of layers and number of neurons you want. For the same reason, we study only feedforward neural networks. In such a network, neurons in this layer were only connected to neurons in the next layer, and they are don't form a cycle. In Feedforward signals travel in only one direction towards the output layer.

Feedback neural networks(the other possible kind of network) contain cycles. Signals travel in both directions by introducing loops in the network. The feedback cycles can cause the network's behavior change over time based on its input. Feedback neural networks are also known as recurrent neural networks.

We approach our network as a two-stage regression or classification model, typically represented by a network diagram (see Figure 6.2.1), This network can be used for both classification and regression. For regression, $K = 1$ and there is only one output unit Y_1 at the top. For K-class classification, there are

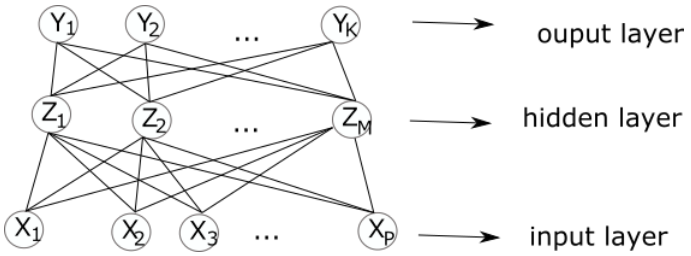


FIGURE 6.2.1. Network diagram

K units at the top (the k-th unit modeling the probability of class k). There are K target measurements Y_k ($k = 1,2,\dots,K$). Derived features Z_m are created from linear combinations of the inputs and, for all k, the target Y_k is modeled as function of a linear combination of the Z_m's. Formally, we have

$$\begin{cases} X = (X_1,\dots,X_P) \\ Z = (Z_1,\dots,Z_M) \\ T = (T_1,\dots,T_K) \end{cases}$$

$$\begin{cases} Z_m = \text{sigm}(\alpha_{0,m} + \alpha_m^T X) & (m = 1,\dots,M) \\ T_k = \beta_{0,k} + \beta_k^T Z & (k = 1,\dots,K) \\ Y_k = g_k(T) \end{cases}$$

with coefficients: $\alpha_{0,m} \in \mathbb{R}$, $\alpha_m \in \mathbb{R}^P$ ($m \in \{1,2,\dots,M\}$), $\beta_{0,k} \in \mathbb{R}$, $\beta_k \in \mathbb{R}^M$ ($k \in \{1,2,\dots,K\}$), and again: sigm is the sigmoid function (in the context of neural networks, it is also called the activation function).

REMARK 6.1. You can add a bias unit feeding into every unit in the hidden and the output layers. This bias will capture the intercepts $\alpha_{0,m}$ and $\beta_{0,k}$ in the above model. With the `neuralnet` command in R, this is what happens. See Figure 6.2.2. We will come back to this example later). Here, the input layer is on the left, the hidden layer is in the middle and the output layer is on the right.

For each k, the output function g_k allows for a final transformation of the vector of outputs. For regression, we choose the identity: $g_k(T) = T_k$. For a K-classification, we choose the soft-max function:

$$g_k(T) = \frac{e^{T_k}}{\sum_{l=1}^K e^{T_l}}$$

(you get positive estimates that sum to one).

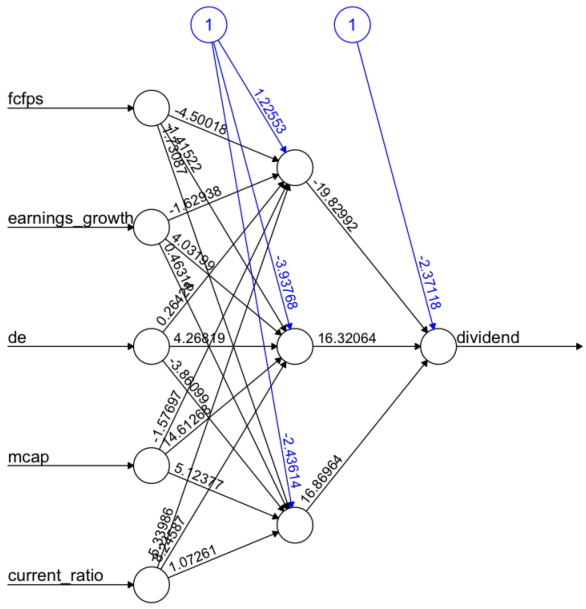


FIGURE 6.2.2. Neural network (with weights).

The units in the middle of the network are called hidden units. In general, there can be more than one hidden layer. It is useful to think of the Z_m as a basis expansion of the original inputs X; the neural network can than be seen as a standard linear model. Here the parameters of the basis on which we expand X (namely, the parameters α's) are learned from the data.

REMARK 6.2. If we replace sigm by the identity then T is a linear function of X (no need of a layer). Hence a neural network is a nonlinear generalization of the linear model (both for regression and classification). If the α's are very small then the sigm in Z_m $(m = 1,2,\ldots,M)$ will operate in its linear part (see Figure 1.4.4, sigm is linear in a neighborhood of zero) and we are back to a linear model.

REMARK 6.3. Initially, the network was used to model parts of the brain and sigm was a step function. Such a step function is not suitable for optimization purposes and so it was replaced by a sigmoid function.

6.3. Fitting neural networks

Unknown parameters are called weights. The complete set of weights is denoted by θ and consists of

$$\begin{array}{ll} \{\alpha_{0,m}, \alpha_m (\in \mathbb{R}^p); m = 1,2,\ldots,M\} & M(p+1) \text{ weights}, \\ \{\beta_{0,k}, \beta_k (\in \mathbb{R}^M); k = 1,2,\ldots,K\} & K(M+1) \text{ weights}. \end{array}$$

For a regression, we use the sum of squared errors as our measure of fit

$$R(\theta) = \sum_{k=1}^{K} \sum_{i=1}^{N} (y_k^{(i)} - f_k(x^{(i)}))$$

$(y^{(i)} = (y_1^{(i)},\ldots,y_K^{(i)})$, $i \in \{1,2,\ldots,N\}$ are the outputs in the training set, $x^{(i)} = (x_1^{(i)},\ldots,x_P^{(i)})$, $i \in \{1,2,\ldots,N\}$ are the inputs in the training set, f_k is the output of the neural network for neuron K) where the total number of points in the training set is N.

For classification, we use either the squared error or the cross-entropy:

$$R(\theta) = -\sum_{i=1}^{N} \sum_{k=1}^{K} y_k^{(i)} \log(f_k(x^{(i)})).$$

Why does it make sense to use the cross-entropy?

REMARK 6.4. We look at ($y \in \mathbb{R}^K$)

$$H(.y) : z \in \mathbb{R}^K \mapsto -\sum_{k=1}^{K} y_k \log(z_k).$$

We want to minimize H under the constrain: $\sum_{k=1}^K y_k = 1$ (all y_k nonnegative), $\sum_{k=1}^K z_k = 1$ (all z_k nonnegative). We compute the gradient of $z \mapsto H(y,z)$

$$\nabla_z H(y,z) = \begin{pmatrix} -y_1/z_1 \\ \vdots \\ -y_n/z_n \end{pmatrix},$$

and we compute the gradient of the constrain

$$\nabla(\sum_{k=1}^{K} z_k) = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}.$$

We are looking for a point where the gradients above a co-linear. This is a point where (for some λ)

$$-\frac{y_k}{z_k} = \lambda \, (\forall k).$$

Under the constrains, it becomes: $z_k = y_k$ ($\forall k$). As H is convex, this critical point is a minimum.

So, what we gather from this Remark is that the parameters such that $R(\theta)$ is minimal should be such that $y_k^{(i)} = f_k(x^{(i)})$ (for all i,k). Maybe there are no parameters such that this equality occurs, but it tells us that minimizing $R(\theta)$ is almost the same thing as tending to $y_k^{(i)} = f_k(x^{(i)})$ (for all i,k) (which is what we want).

To minimize $R(\theta)$, we use a gradient descent (something we have seen before). In the setting of neural networks, this particular step is called back-propagation.

Let us have a look at the computation in details in the case of the squared error loss. We have

$$\begin{cases} z_m^{(i)} &= \text{sigm}(\alpha_{0,m} + \alpha_m^T x^{(i)}) \\ z^{(i)} &= (z_1^{(i)}, \dots, z_M^{(i)}). \end{cases}$$

I can then write

$$R(\theta) = \sum_{i=1}^{N} \underbrace{\sum_{k=1}^{K} (y_k^{(i)} - f_k(x^{(i)}))^2}_{=:R_i}.$$

I compute derivatives

$$\begin{cases} \frac{\partial R_i}{\partial \beta_{k,m}} &= \underbrace{-2(y_k^{(I)} - f_k(x^{(i)}))g_k'(\beta_k^T z^{(i)})}_{=:\delta_{k,i}} \times z_m^{(i)} \\ \frac{\partial R_i}{\partial \alpha_{m,l}} &= \underbrace{-\sum_{k=1}^{K} 2(y_k^{(i)} - f_k(x^{(i)}))g_k'(\beta_k^T z^{(i)})\beta_{k,m} \text{sigm}'(\alpha_m^T x^{(i)})}_{=:s_{m,i}} x_l^{(i)}. \end{cases} \tag{6.1}$$

The gradient descent update a the $(r+1)$-th iteration has the form

$$\begin{cases} \beta_{k,m}^{(r+1)} &= \beta_{k,m}^{(r)} - \gamma_r \sum_{i=1}^N \frac{\partial R_i}{\partial \beta_{k,m}}(\beta_{k,m}^{(r)}) \\ \alpha_{m,l}^{(r+1)} &= \alpha_{m,l}^{(r)} - \gamma_r \sum_{i=1}^N \frac{\partial R_i}{\partial \alpha_{m,l}}(\alpha_{m,l}^{(r)}) \end{cases}$$

where γ_r is the learning rate. We can rewrite Equation (6.1) as

$$\begin{cases} \frac{\partial R_i}{\partial \beta_{k,m}} &= \delta_{k,i} z_m^{(i)} \\ \frac{\partial R_i}{\partial \alpha_{m,l}} &= s_{m,i} x_l^{(i)} \end{cases}$$

We can write

$$s_{m,i} = \overset{\prime}{\text{sigm}}(\alpha_m x^{(i)}) \sum_{k=1}^{K} \beta_{k,m}\delta_{k,i}$$

($1 \le i \le N$, $1 \le m \le M$). The formula for the δ's (previous session) includes the y's, z's. From the δ's, we compute the s's. This equation is called the back-propagation equation.

Each update in the gradient descent is made of two steps (two-pass algorithm).

- Forward pass: The weights ($\alpha_{\dots}$ and $\beta_{\dots}$) are fixed and the predicted values $f_k(x)$ are computed using equation (... N1, at the beginning of the notes ...).
- Backward pass: The errors $\delta_{\dots}$ are computed and then back-propagated to compute the coefficients $s_{\dots}$.

Once you have the coefficients (/errors) $\delta_{\dots}$ and $s_{\dots}$, you can compute the gradient update (you compute $\theta^{(r+1)}$ from $\theta^{(r)}$.

Complexity of the scheme: You can imagine the complexity is big because each time you do a forward or backward pass, each unit/neuron receives information from units/neurons which is connected to it (example: the cost of a forward pass is of order $M \times P + M \times K$). This not good but this algorithm can be implemented efficiently on parallel architecture computer (most laptops have more than one core and also have a graphical unit, so you can parallelize your code inside your own computer).

Notation: Here, we used all the training data, this is called batch-learning to update the parameters. We could use part of the training data (mini-batch gradient descent) or only one point in the data (stochastic gradient descent: batch-size=1). We would like to cycle through the data. One sweep through the entire training set is called a training epoch.

Learning rate: The learning rate γ_r for batch learning is usually taken to be a constant. It can be optimized by a line search that minimizes the error function at each update.

6.4. Some issues in training neural networks

6.4.1. Starting values. You choose the weights $\alpha_{\dots}$, $\beta_{\dots}$ to be random, around zero.

- If the weights are all zero: they will remain at zero forever.
- If the weights are too large, the algorithm will not converge (or very slowly) and, as you have to stopit at some point, you will get very poor solutions.

6.4.2. Over-fitting.

EXAMPLE 6.5. You want teach your network to label pictures as "cat" or "dog". Imagine, in your data set, you have a lot of pictures of dogs including also a chair. After training your network, it will label "dog" any picture in which it sees a chair.

REMARK 6.6. When you train your model, you split the data into a training set and a test set. You pick parameters for your model (in our case, choose the weights $\alpha_{\dots}$, $\beta_{\dots}$). Then you can compute the error rate on the test set. You can also compute the error on the training set (you apply your classifier to the data and compute the failure rate/success rate). If, for example, you get a success rate of 98% on the training set and 89% on the test set, then you are clearly over-fitting.

The usual response to the over-fitting problem is to use some kind of regularization. In the neural network case, it is called weight decay. We add a penalty to the error function: replace $R(\theta)$ by $R(\theta) + \lambda J(\theta)$, where

$$J(\theta) = \sum_{k,m} \beta_{k,m}^2 + \sum_{m,l} \alpha_{m,l}^2$$

(λ is the tuning parameter). Typically, cross-validation is used to estimate λ.

Another form of penalty:

$$J(\theta) = \sum_{k,m} \frac{\beta_{k,m}^2}{1+\beta_{k,m}^2} + \sum_{m,l} \frac{\alpha_{m,l}^2}{1+\alpha_{m,l}^2}$$

(weight elimination penalty).

6.4.3. Scaling of the inputs. It is best to standardize all inputs to have mean zero and deviation one. It is typical (with such standardized inputs) to initialize the weights as uniform variables in $[-0.7, +0.7]$.

6.4.4. Number of hidden units and hidden layers. Choice of the numbers of hidden units and layers is guided by background knowledge and experimentation. In any case, you should experiment many configurations to find which one suits the best your problem.

You can add layers, following the intuition that a deep model (e. g. one with a lot of layers) provides a hierarchy of layers that build up increasing levels of abstraction from the space of the input variables to the output variables.

6.4.5. Multiple mining. The error function $R(\theta)$ is non-convex, possessing many local minima. As a result, the final solution obtained is quite dependent on the choice of starting weights. One must at least try a number of random starting configurations and choose the solution giving the lowest (penalized) error.

- One can use the average predictions over the collection of networks you have created as the final prediction.
- Averaging the weights is a bad idea since the non-linearity of the model implies that this averaged solution can be quite bad.

6.5. Advantages/disadvantages of neural networks

Advantages.

- Processing vague, incomplete data.
- Effective at recognizing patterns (in images).
- Parallel processing ability: Artificial neural networks have numerical strength that can perform more than one job at the same time.

Disadvantages.

- Black box: One of the most distinguishing disadvantages of the neural network is their ''black box" nature. It means that we don't know how and why the neural network came up with a certain output. (`https://www.smbc-comics.com/comic/simulation-2`)
- Amount of data: Neural networks require much more data than any other traditional machine learning algorithms.
- Determination of proper network structure: There is no specific rule for determining the structure of a neural network. The appropriate network structure is achieved through experience and trial and error.
- Computationally Expensive.

6.6. Examples in R

6.4.1. Dividends. We download the `dplyr` package (useful for data manipulation, it is needed for the `select` command used below).

```
library(dplyr)
```

We download the data[1].

```
myData<-read.csv("dividendinfo.csv")
head(myData)
```

Each row of the data set (or observation) is relative to a stock. We have five input variables:

- `fcfps`: Free cash flow per share (in $)
- `earnings_growth`: Earnings growth in the past year (in %)
- `de`: Debt to Equity ratio
- `mcap`: Market Capitalization of the stock
- `current_ratio`: Current Ratio (or Current Assets/Current Liabilities)

[1] `https://github.com/MGCodesandStats/datasets/blob/master/dividendinfo.csv`

The output variable (`dividend`) is made in the following way. In our dataset, we assign a value of 1 to a stock that pays a dividend. We assign a value of 0 to a stock that does not pay a dividend. In this particular example, our goal is to develop a neural network to determine if a stock pays a dividend or not

We split the data into a training set and a test set

```
trainSet<-myData[1:160,]
testSet<-myData[161:200,]
```

Let us say we want to train a neural network with two hidden layers (with, respectively, three and two neurons). We want to make sure we are not over-fitting. We use the `neuralnetwork` command.

```
library(ANN2)
NN<-neuralnetwork(trainSet[,-1],trainSet[,1],hidden.layers=c(3,2),
        standardize = TRUE,n.epochs = 1500)
```

Here are a few comments about this command.

- The two first arguments are the input variables and the output variable.
- We choose the structure of the network with `hidden.layers=c(3,2)` (here we have the input layer, then a layer with three neurons, then a layer with two neurons, then the output layer).
- The option `standardize=TRUE` standardizes the data.
- When can choose how long we want to run the fitting algorithm with the option `n.epochs`.

We plot the loss (as a function of the number of epochs) (see Figure Here, we see that R has split the

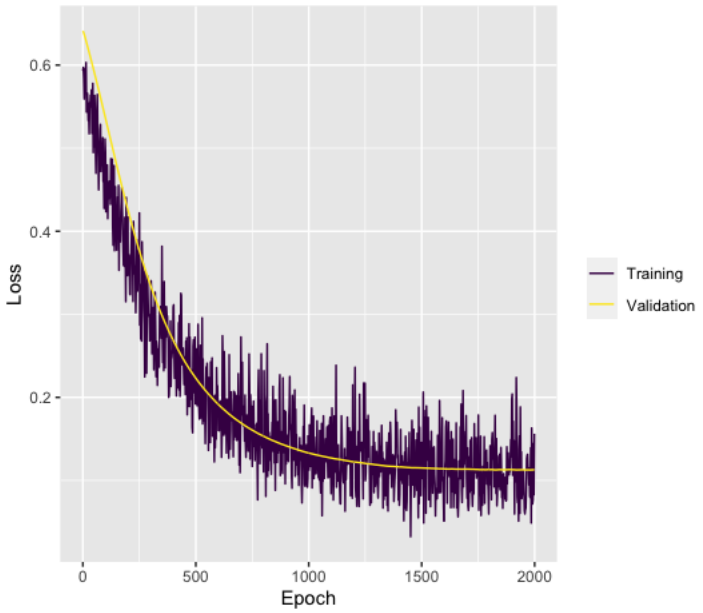


FIGURE 6.6.1. Loss

training set into a smaller training set and a validation set. The losses for the small training set and the validation set are similar. The way to choose `n.epochs` is to take it large enough to see the loss converging towards something. Caution: the data set is small so you will not get exactly the same plot each time you run your program. But, in average, the two curves are at the same height. Some other times, we get a plot similar to the one of Figure 6.6.2 So we decide, there is some over-fitting there. We can use the weight decay we have seen in Section 6.4.2. This is done with the option `L2=0.5`.

```
NN<-neuralnetwork(trainSet[,-1],trainSet[,1],hidden.layers=c(3,2),
                                L2=0.5,standardize = TRUE,n.epochs = 2000)
plot(NN)
```

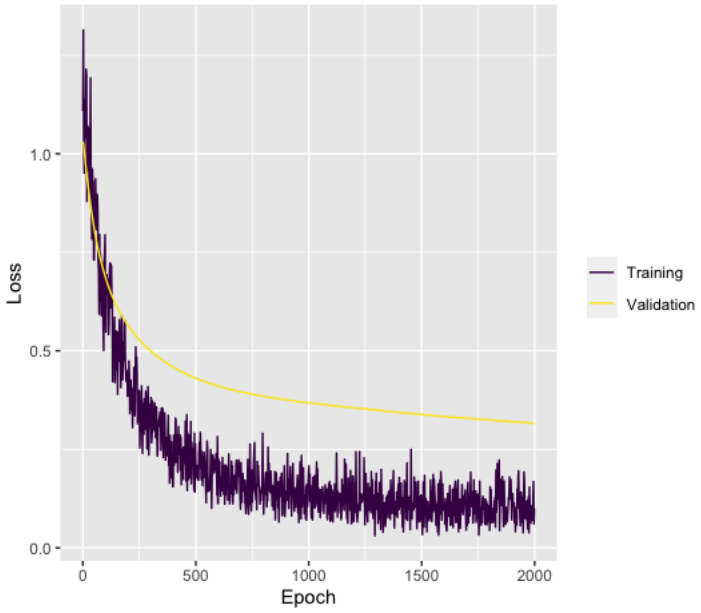


FIGURE 6.6.2. Over-fitting

And we get the graph in Figure 6.6.3 (again, the result is random, so what happens is that we get this

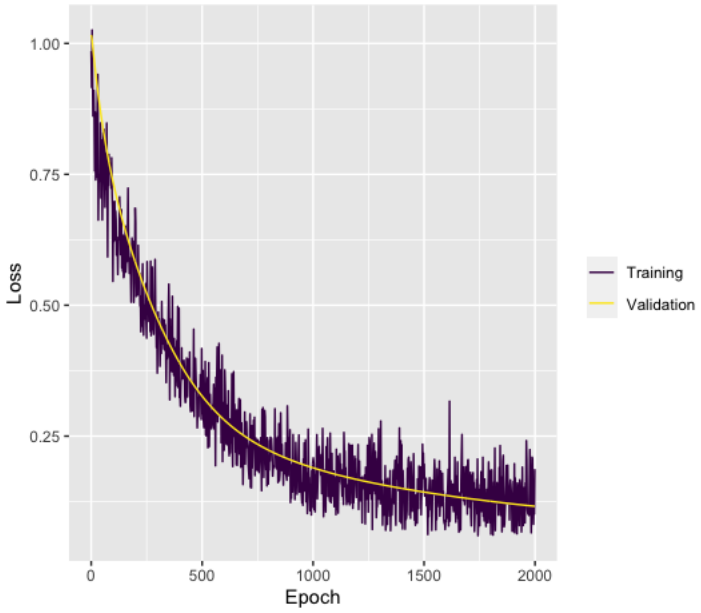


FIGURE 6.6.3. No more over-fitting

graph more often than before). So we consider we do not have an over-fitting problem anymore. The parameter `L2=...` has to be hand-picked so that the two curves coincide.

The las step is the prediction.

```
pred<-predict(NN, newdata=testSet[,-1])
table(pred$predictions, testSet[,1])
cm
```

```
library(caret)
ccm<-confusionMatrix(cm)
cmm$overall[1]
```

The confusion matrix is

```
   0  1
0 17  4
1  0 19
```

and the accuracy is 89.7%.

Let us now explore what we can do with the `neuralnet` command. This command allows you to train a neural network. It does not allow to implement the weight decay or the data scaling but it has other nice options.

We want to scale the input data using the `scale` function. So we remove the `dividend` column before using `scale`, and then we replace it in the data. The `scale` function substract to each column the empirical mean (of each column) and then divides each column by the standard deviation (of each column). This is a nice feature of `R` that avoids us to think about how we want to center and scale our data.

```
library(dplyr)
scaledTrain<-trainSet %>%
                subset(select=-dividend) %>%
                 scale
trainSet<-cbind(dividend=trainSet$dividend,scaledTrain)
```

⚠ We then want to re-scale the test data, using the *same mean and standard deviation* (and not use the `scale` command with default options again).

```
scaledTest<-testSet %>%
        subset(select=-dividend) %>%
         scale(center=attr(scaledTrain,"scaled:center"),
        scale=attr(scaledTrain,'scaled:scale'))
testSet<-         cbind(dividend=testSet$dividend,scaledTest)
```

We use the package `neuralnet` and the command `neuralnet` to train a neural network.

```
library(neuralnet)
optimNet <- neuralnet(dividend ~ fcfps + earnings_growth + de + mcap
                                                    + current_ratio,
                      data=trainSet, hidden=c(3,2), linear.output=FALSE,
                      threshold=0.01)
```

Here are a few comments about the `neuralnet` command.

- The option `linear.output= FALSE` introduces the sigmoid activation function in the output neurons, this function is by default present in the intermediate layers. In other words, the argument `linear.output` is used to specify whether we want to do regression (`linear.output=TRUE`) or classification (`linear.output=FALSE`).
- For some reason the formula ''`y~.`'' is not accepted in the `neuralnet()` function. You need to first write the formula and then pass it as an argument in the fitting function. This could be difficult if we had a lot a variables. Here is a smart way to solve this problem

  ```
  var <-paste(colnames(subset(trainSet,select=-dividend)),collapse="+")
  formule<-paste("dividend ~ ",var,sep="")
  optimNet <- neuralnet(formule, data=trainSet, hidden=c(3,2),
                        linear.output=FALSE, threshold=0.01)
  ```

- The `threshold` parameter is a numeric value specifying the threshold for the partial derivatives of the error function as stopping criteria. In doubt, let `R` choose a default value.

- The `stepmax` parameter is the maximum steps for the training of the neural network. Reaching this maximum leads to a stop of the neural network's training process.
- The `act.fct=...` option is the choice of the activation function (default=sigmoid).
- The option `hidden=c(3,2)` specifies the shape of the network. Here we get the network of Figure 6.6.4

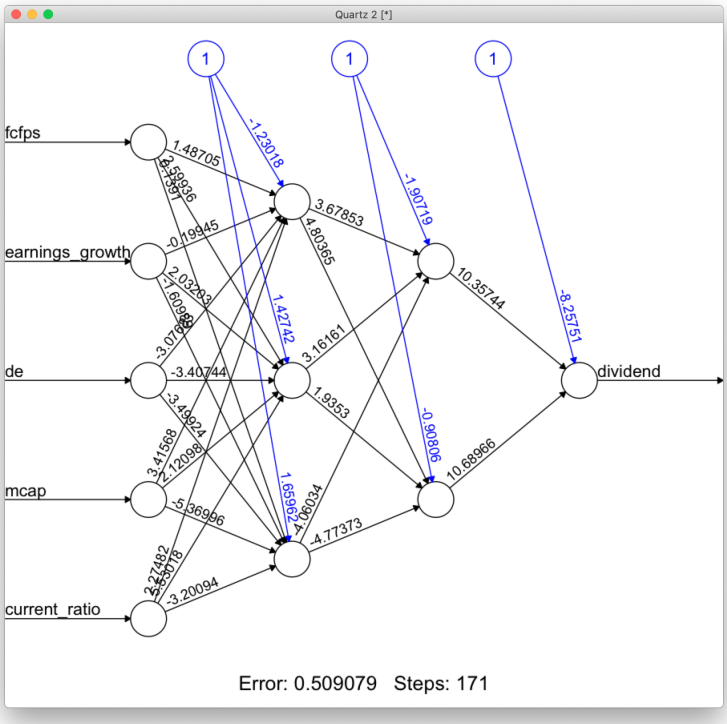


FIGURE 6.6.4. Neural network

We can see the parameters chosen using

```
optimNet$result.matrix
```

(we get a long list of numbers) or using

```
plot(optimNet)
```

which gets us Figure 6.6.4 (we can see that the computer needed 171 steps to train the network, the error is ... something calculated by the computer but we will not get into it).

Now, we can test the resulting output. We get rid of the `dividend` column

```
temp_test <- subset(testSet, select = c("fcfps","earnings_growth", "de", "mcap",
"current_ratio"))
head(temp_test)
```

We predict using `predict`

```
optimNet.results <- predict(optimNet, temp_test)
```

The results are probabilities (for each observation: the probability that the stock pays a dividend. We round these probabilities to 0 or 1

```
optimNet.results <- round(optimNet.results)
```

We can then view the confusion matrix on the test set and compute the accuracy

```
cm<-table( testSet[,1], optimNet.results)
cm
confusionMatrix(cm)$overall[1]
```

The accuracy is 87.5%

After playing a little with the `hidden=c()` option, the best configuration we found is `c(2,1)` (with an accuracy of 92.5%). The accuracy goes to 100% when the number of neurons (or layers) goes to infinity but this does not mean that a simpler configuration cannot have better results than a complicated one. We observe also that `hidden=c(5)` (only one layer but with more neurons) gets is an accuracy of 80.15% only. So there is an obvious gain in adding layers.

6.6.2. Bank marketing data set. The data is related with direct marketing campaigns (phone calls) of a Portuguese banking institution. The classification goal is to predict if the client will subscribe a term deposit (variable y). The data comes from `http://archive.ics.uci.edu/ml/datasets/Bank+Marketing` and is discussed widely on the internet. The file you have to load is `bank.csv`. The full description of the data is the following

(1) - age (numeric)
(2) - job : type of job (categorical: 'admin.','blue-collar','entrepreneur','housemaid','management', 'retired','self-employed','services','student','technician','unemployed','unknown')
(3) - marital : marital status (categorical: 'divorced','married','single','unknown'; note: 'divorced' means divorced or widowed)
(4) - education (categorical: 'basic.4y','basic.6y','basic.9y','high.school','illiterate', 'professional.course','university.degree','unknown')
(5) - default: has credit in default? (categorical: 'no','yes','unknown')
(6) - housing: has housing loan? (categorical: 'no','yes','unknown')
(7) - loan: has personal loan? (categorical: 'no','yes','unknown') # related with the last contact of the current campaign:
(8) - contact: contact communication type (categorical: 'cellular','telephone')
(9) - month: last contact month of year (categorical: 'jan', 'feb', 'mar', ..., 'nov', 'dec')
(10) - day_of_week: last contact day of the week (categorical: 'mon','tue','wed','thu','fri')
(11) - duration: last contact duration, in seconds (numeric). Important note: this attribute highly affects the output target (e.g., if duration=0 then y='no'). Yet, the duration is not known before a call is performed. Also, after the end of the call y is obviously known. Thus, this input should only be included for benchmark purposes and should be discarded if the intention is to have a realistic predictive model.
(12) - campaign: number of contacts performed during this campaign and for this client (numeric, includes last contact)
(13) - pdays: number of days that passed by after the client was last contacted from a previous campaign (numeric; 999 means client was not previously contacted)
(14) - previous: number of contacts performed before this campaign and for this client (numeric)
(15) - poutcome: outcome of the previous marketing campaign (categorical: 'failure','nonexistent', 'success') # social and economic context attributes

We download the data

```
bank_raw<-read.csv("bank/bank.csv",sep=";")
```

(wee nee to specify the separator, see by yourself what happens if we do not). We assign the number 0 if education level is "primary", 1 if the education level is "secondary", 2 if the education level is "tertiary").

```
bank_raw$education_num<-ifelse(bank_raw$education=="primary",0,
                               ifelse(bank_raw$education=="secondary",1,2))
bank_raw$education<-NULL
```

We transform the output data into numerical.

```
bank_raw$y<-ifelse(bank_raw$y=="yes",1,0)
```

We get rid of the `day` column.

```
bank_raw$day<-NULL
```

We shuffle the data (so that it will be well distributed when we split it into a training set and a test set).

```
rows<-sample(nrow(bank_raw))
bank<-bank_raw[rows,]
```

Some categorical variables cannot be represented easily as numerical variables. We create dummy variables from these (see Section 2.8).

```
bank_temp<-select(bank,job,marital,default,housing,loan,contact,poutcome,month)
library(caret)
du<-dummyVars("~.",data=bank_temp,fullRank=TRUE)
library(data.table)
du_table<-data.table(predict(du,bank_temp))
bank_table<-select(bank,-job,-marital,-default,-housing,
                                        -loan,-contact,-poutcome,-month)
bank_num=cbind(du_table,bank_table)
```

Some column names might be a problem later so we transform them.

```
colnames(bank_num)[1]<-"jobbluecollar"
colnames(bank_num)[6]<-'jobselfemployed'
```

You would get the following message later if you do not do this transformation.

```
Error in `[.data.frame`(data, , model.list$variables):undefined columns selected
```

We split the data into a training set and a test set (we use the `data.frame` command to get data-frames and not matrices). The `dplyr` package is needed to use the pipe operator `%>%`.

```
library(dplyr)
bankTrain<-bank_num[1:3200,] %>% data.frame
bankTest<-bank_num[3201:4521,]  %>% data.frame
```

We scale the training set variables (but do not scale the output variable.

```
bankTrainInput <- bankTrain %>% subset( select=-y) %>% scale
```

⚠We scale the test set variables using the *same mean and standard deviation* (and not use the `scale` command with default options again).

```
bankTestInput <- bankTest %>%
                                    subset( select=-y) %>%
scale(center=attr(bankTrainInput,"scaled:center"),
                  scale=attr(bankTrainInput,'scaled:scale'))
```

We bind the output variable to the data-frames.

```
bankTrain <- cbind(bankTrainInput,y=bankTrain$y) %>% data.frame
bankTest <- cbind(bankTestInput,y=bankTest$y)          %>% data.frame
```

As before, we define the formula we want to use for the training of our neural net.

```
col_list<-paste(c(colnames(select(bankTrain,-y))),collapse='+')
col_list<-paste("y ~ ",col_list,collapse="")
formule<-formula(col_list)
```

We define and train our neural network (two hidden layers with ten neurons each)

```
library(neuralnet)
nmodel<-neuralnet(      formule,data=bankTrain,hidden=c(10,10),
   threshold = 0.01, stepmax=1e+06,linear.output=FALSE)
```

We can try our neural net on our test set. The neural net will predict probability (probability that the customer subscribes a term deposit) so we round it to 0 or 1 .

```
tempTest<-subset(bankTest, select=-y)
nmodel.results <- predict(nmodel, tempTest)
nmodel.results <-round(nmodel.results)
```

We visualize our results (confusion matrix and accuracy).

```
cm<-table(bankTest$y, nmodel.results)
cm
cmm<-confusionMatrix(cm)
cmm$overall[1]
```

We get an accuracy of 85.7%. On most websites where people explain how they used a neural net to predict the outcome for this particular data set, these people find this result satisfactory. Let us compute the percentage of variable `y` being "`yes`" in our data.

```
sum(bank_num$y) / nrow(bank_num)
```

This percentage is 11.5%. This means that if my prediction is, every single time: "the customer will not subscribe a term deposit", I will be right 88.5% of the time (so my accuracy is 88.5%). So our neural net performs very poorly indeed.

Of course, we have to try with more data (data set `bank-full.csv`) and with different configurations (more neurons, more layers). The author of this handout has tried and the result is always similar.

So we have to question our whole strategy here. Remember neural nets are made to perform tasks a human could perform (sometimes the human would be slower). If you look at the input variables (job, marital status, ...), it is very dubious that you could use them to predict the outcome of a marketing phone call. Some other variables would be useful (how gullible this person is, etc) but we do not have access to them. The conclusion is that the whole enterprise was foolish from the beginning.

6.6.3. House price (activity). We use here the data set[2] `housepricedata.csv`. This is about home value and the task is to predict whether the house price is above or below median value. We have our input features in the first ten columns:

- Lot Area (in sq ft)
- Overall Quality (scale from 1 to 10)
- Overall Condition (scale from 1 to 10)
- Total Basement Area (in sq ft)
- Number of Full Bathrooms
- Number of Half Bathrooms
- Number of Bedrooms above ground
- Total Number of Rooms above ground
- Number of Fireplaces Garage Area (in sq ft)

In our last column, we have the feature that we would like to predict:

- Is the house price above the median or not? (1 for yes and 0 for no)

Questions:

- Split your data into a train set and a test set.
- Check whether there is a tendency to over-fitting in this model, using `neuralnetwork`.
- Train your network (for various settings) and test its predictions on the test set.

Answer : you should get an accuracy above 80%. Playing with the number of layers and number of neurons, you could get above 90%.

6.6.4. Summary of useful `R` commands. The `neuralnetwork` command (`library(ANN2)`). It has the following form.

```
library(ANN2)
NN<-neuralnetwork(trainSet[,-1],trainSet[,1],hidden.layers=c(3,2),
        standardize = TRUE,n.epochs = 1500)
```

[2] `https://www.freecodecamp.org/news/how-to-build-your-first-neural-network-to-predict-house-prices-with-keras-f8db83049`

Here are a few comments about this command.

- The two first arguments are the input variables and the output variable.
- We choose the structure of the network with `hidden.layers=c(3,2)` (here we have the input layer, then a layer with three neurons, then a layer with two neurons, then the output layer).
- The option `standardize=TRUE` standardizes the data.
- When can choose how long we want to run the fitting algorithm with the option `n.epochs`.
- Plot the loss function : `plot(NN)`
- Make a prediction with `pred<-predict(NN,newdata=...)`.

The `scale` function: it scales data (removes the empirical mean and divide by the empirical standard deviation). Example:

```
scaledData<-scale(data)
```

Access to the mean and standard deviation:

```
mean<-attr(scaledData,"scaled:center"))
sd<-attr(scaledData,'scaled:scale') )
```

The `neuralnet` command. Example:

```
library(neuralnet)
optimNet <- neuralnet(dividend ~ fcfps + earnings_growth + de + mcap
                                                    + current_ratio,
                      data=trainSet, hidden=c(3,2), linear.output=FALSE,
                      threshold=0.01)
```

- The option `linear.output= FALSE` introduces the sigmoid activation function in the output neurons, this function is by default present in the intermediate layers. In other words, the argument `linear.output` is used to specify whether we want to do regression (`linear.output=TRUE`) or classification (`linear.output=FALSE`).
- For some reason the formula ‘‘`y~.`’’ is not accepted in the `neuralnet()` function. You need to first write the formula and then pass it as an argument in the fitting function. This could be difficult if we had a lot of variables. Here is a smart way to solve this problem

  ```
  var <-paste(colnames(subset(trainSet,select=-dividend)),collapse="+")
  formule<-paste("dividend ~ ",var,sep="")
  optimNet <- neuralnet(formule, data=trainSet, hidden=c(3,2),
                        linear.output=FALSE, threshold=0.01)
  ```

- The `threshold` parameter is a numeric value specifying the threshold for the partial derivatives of the error function as stopping criteria. In doubt, let `R` choose a default value.
- The `stepmax` parameter is the maximum steps for the training of the neural network. Reaching this maximum leads to a stop of the neural network's training process.
- The `act.fct=...` option is the choice of the activation function (default=sigmoid).
- The option `hidden=c(3,2)` specifies the shape of the network. Here we get the network of Figure 6.6.4
- We can see the parameters chosen using `optimNet$result.matrix` or get a nice plot of the network using `plot(optimNet)`.
- Make a prediction with `pred<-predict(NN,testData)`.

CHAPTER 7

Tree based methods

In this chapter, we are going to talk about tree-based methods for regression and classification (an instance of supervised learning). These methods imply segmenting or stratifying the input space (the space in which we have input points $x^{(1)}, \ldots, x^{(N)}$, as usual in this handout, this space is $\mathbb{R}^D$) into a number of simple regions. When we want to make a prediction for a given observation, we typically use the mean or the mode of the training observations in the region to which it belongs. The set of splitting rules used to segment the space can be summarized in a tree, so these methods are called decision tree methods.

One advantage of these methods is that trees are useful for interpretations. On the downside, they are not usually competitive with the other supervised learning methods, in terms of prediction accuracy. But, as we will see later in this Chapter, some methods called bagging and random forests can improve the accuracy. These approaches involve producing multiple trees and combine them to get a single prediction. This will get us improvements in prediction accuracy, at the expense of some loss in interpretability. We will not talk about boosting in this Chapter (another improvement over decision trees), the curious reader can have a look at [**GWHT17**], Section 8.2.3, p. 221 for a clear explanation on boosting.

7.1. Decision trees

Decision trees can be applied to both regression and classification. We will first present regression problems because they are simpler. These methods are also known as CART (Classification and Regression Trees).

7.1.1. Regression trees. We begin with a simple example.

Predicting baseball players' salary using regression trees. We use the `Hitters` data set (comes with `library(IRLS)`) to predict a baseball player's `Salary` given: `Years` (the number of years he has played in the major leagues) and `Hits` (the number of hits he made in the previous year). We remove observations with missing `Salary` and take the logarithm of `Salary` (`Salary` is measured in thousands of dollars).

In Figure 7.1.1, we can see a regression tree fit to this data. The tree is made of a series of splitting

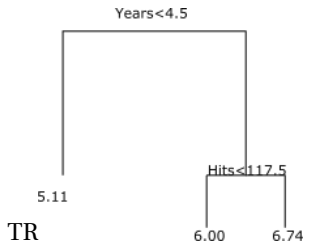


FIGURE 7.1.1. Regression tree

rules, starting at the top of the tree. If `Years<4.5`, we go down the left branch. The predicted salary is then given by the mean value for the players of the dataset with `Years<4.5`. For such players, the mean logarithm of the salary is 5.107, so our prediction is $e^{5.11}$ thousands of dollars, i.e. \$165,670. If `Years>4.5`, we go down the right branch, and then if `Hits<117.5`, the prediction for the logarithm of the salary is 6.00, and it is 6.74 in the other case. Overall, the tree segments the players into three regions:

- players who have played four or fewer years (`Years` takes integer values),
- players who have played five years or more and made fewer than 118 hits last year,
- players who have played five years or more and made at least 118 hits last year.

These three regions can be written

- $R_1 = \{X | \texttt{Years<4.5}\}$,
- $R_2 = \{X | \texttt{Years>4.5}, \texttt{Hits<117.5}\}$,
- $R_3 = \{X | \texttt{Years>4.5}, \texttt{Hits>117.5}\}$,

where X is a generic symbol for points of coordinates (`Years`,`Hits`). We can see a drawing of these regions in Figure 7.1.2.

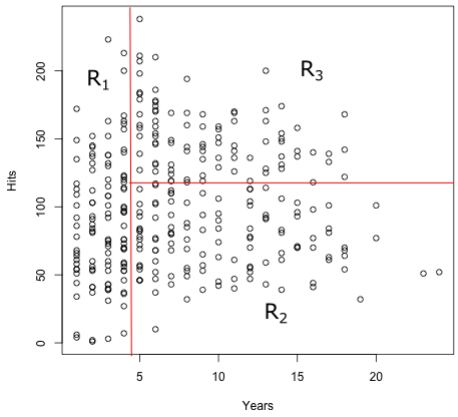


FIGURE 7.1.2. The three-regions partition for the `Hitters` data set from the regression tree illustrated in Figure 7.1.1

The regions R_1, R_2 and R_3 are called leaves (or terminal nodes). The decision trees are usually drawn upside down with the leaves at the bottom of the tree. The points along the tree where there is a split into two branches are called internal nodes. In Figure 7.1.1, the internal nodes are indicated by the text `Years<4.5` and `Hits<117.5`. The segments of the trees that connect the nodes are called branches.

We interpret the regression tree of Figure 7.1.1 as follows: `Years` is the most important factor in determining `Salary`, and players with less experience earn lower salaries than more experienced players. Given that a player is more experienced, the numbers of hits that he made in the previous year plays a role in his salary (players with more hits tend to have higher salaries). This regression tree is an over-simplification for the true relationship between `Hits`, `Years` and `Salary`. However, it has the advantage to be easy to interpret.

Prediction via stratification of the space. We are now going to explain how we build a regression tree. There are roughly two steps.

(1) Divide the space into J distinct and non-overlapping regions $R_1, \ldots, R_J$.
(2) For every input that falls into the region R_j, we make the same prediction: the mean of the outputs for the training points in R_j.

For example, suppose that, in Step 1, we obtained two regions R_1, R_2, and that the output mean of the training observations in the first region is 10 , while the output mean of the training observation is 20 in the second region. Then, for a new input x, if x is in R_1, we will predict a value of 10, and if x is in R_2, we will predict a value of 20.

We are now going to explain how we build the regions $R_1, \ldots, R_J$. The regions could have any shape but we decide to divide the space into high-dimensional rectangles, or boxes. This will get us simple and easily interpretable predictions. The goal is to find boxes $R_1, \ldots, R_J$ that minimizes the RSS (residual

sum of squares) given by

$$\sum_{j=1}^{J} \sum_{i \in R_j} (y^{(i)} - \overline{y}_{R_j})^2 ,$$

where $\overline{y}_{R_j}$ is the mean of the training outputs within the j-th box (the outputs are $y^{(1)}, \dots, y^{(N)}$). It is not computationally feasible to consider all possible partitions of the space into J boxes. For this reason, we resort to a top-down approach(or greedy approach) that is known as recursive binary splitting . This is a top-down approach because it begins at the top of the tree (at the begin, all the points belong to a single region) and then successively splits the space; each split is indicated via two new branches further down on the tree. It is greedy because at each step of the process, the best split is made (rather than looking ahead and picking a split that will lead to a better tree in some future step).

In order to perform recursive binary splitting, we first select the feature j and the cut-point s such that splitting the space into the regions $\{x : x_j < s\}$ and $\{x : x_j \geq s\}$ leads to the greatest possible reduction in RSS. So we consider all features (or coordinates) $j = 1, 2, \dots, D$ and all possible values of the cut-point s for each j, and then choose j and s such that the resulting tree has the lowest RSS. We can write this using mathematical formulas: for any j and s, we define

$$R_1(j,s) = \{x : x_j < s\} , R_2(j,s) = \{x : x_j \geq s\} ,$$

and we week the value of (j,s) that minimizes the sum

$$\sum_{i : x^{(i)} \in R_1(j,s)} (y^{(i)} - \overline{y}_{R_1(j,s)})^2 + \sum_{i : x^{(i)} \in R_2(j,s)} (y^{(i)} - \overline{y}_{R_2(j,s)})^2 , \tag{7.1}$$

where $\overline{y}_{R_1(j,s)}$ is the mean output for the training observations in $R_1(j,s)$ and $\overline{y}_{R_2(j,s)}$ is the mean output for the training observations in $R_2(j,s)$. Finding the (j,s) that minimizes (7.1) can be done quite quickly, especially when the number of features D is not too large.

Afterwards, we repeat the process, looking for the best feature and best cut-point in order to split the data further so as to minimize the RSS within each of the resulting regions. Again, we look to split one of these four regions further, so as to minimize the RSS. The process continues until a stopping criterion is reached; for example, we may continue until no region contains more than five observations.

Once the regions $R_1, \dots, R_J$ are created, we predict the response for a given input using the mean of the training outputs in the region to which that test observation belongs.

We can see a five-region example of this approach in Figure 7.1.3.

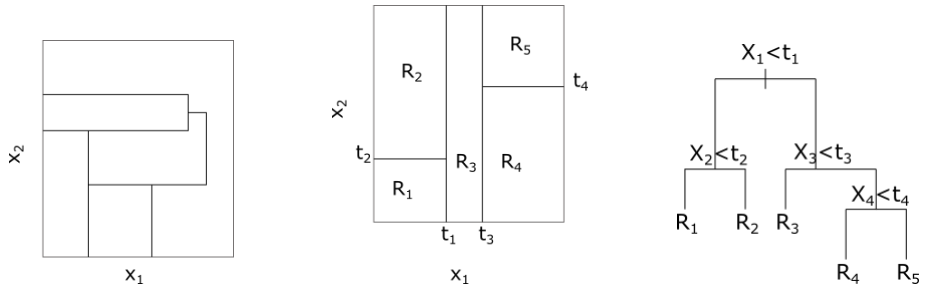


FIGURE 7.1.3. Left: this partition cannot be the result of a recursive binary splitting. Middle: example of a result of a recursive binary splitting. Right: A tree corresponding to the partition in the middle.

Tree pruning. The process described above will produce good predictions on the training set, but it is likely to over-fit the data, leading to poor test set performance. The reason for this is that the resulting tree might be too complex. A smaller tree with fewer splits (and thus fewer regions $R_1, \dots, R_J$) might lead to lower variance and better interpretation at the cost of a little bias.

We are going to grow a very large tree T_0, and then prune it back in order to obtain a subtree. Given a subtree, we can estimate its test error using cross-validation or the validation set (/test set) approach. However, estimating the error for every possible subtree would be too costly since there is an

extremely large number of possible subtrees. Instead, we need a way to select a small set of subtrees for consideration.

We are going to use what is called cost complexity pruning, also known as weakest link pruning. Rather than looking at every possible subtree, we consider a sequence of trees indexed by a nonnegative tuning parameter α. For each α, there corresponds a subtree $T_\alpha \subset T_0$ such that

$$\sum_{m=1}^{|T_\alpha|} \sum_{i\,:\,x^{(i)}\in R_m} (y^{(i)} - \overline{y}_{R_m})^2 + \alpha |T_\alpha| \tag{7.2}$$

is as small as possible, where $|T_\alpha|$ is the number of leaves of the tree T_α, R_m is the box corresponding to the m-th tree and $\overline{y}_{R_m}$ is the mean of the training outputs in R_m. The tuning parameter α creates a trade-off between the subtree's complexity and its fit to the training data. When $\alpha = 0$, the subtree T_α is equal to the original tree T_0 because (7.2) just measures the training error. As α increases, there is a price to pay for having a tree with many leaves, and so the quantity (7.2) will tend to be minimized for a smaller subtree. This is all very similar to the LASSO from Section 2.5, in which a similar penalization was used to control the complexity of a linear model.

When we increase α from 0, branches get pruned from the tree in a nested a predictable fashion (meaning we have a sequence trees, nested in one another) and obtaining the whole sequence of subtrees as a function of α is easy (one can look for a proof of this claim in [**Rip96**], Section 7.2, p. 221). We can then select a value of α using a validation set and obtain the subtree corresponding to α. So the steps to be taken are the following.

(1) Use recursive binary splitting to grow a large tree on the training data, stopping only when each leave (a.k.a. terminal node) has fewer than some minimum number of points.
(2) Apply cost complexity pruning to the large tree in order to obtain a sequence of best subtrees, as a function of α.
(3) Use K-fold cross-validation to choose α: divide the training observations into K folds, for each $k = 1, \dots, K$, do
    (a) Repeat steps 1 and 2 on all but the k-th fold of the training data.
    (b) Evaluate the mean squared prediction error on the data in the left-out k-th fold, as a function of α.

    Average the results for each value of α, and pick α to minimize the average error.
(4) Return the subtree from step 2 that corresponds to the chosen value of α.

In Figure 7.1.4, we can see the result of fitting a regression tree on the `Hitters` data, using all the features. We have split the data into a training set (with 132 observations) and a test set (with 131 observations) (beforehand, we removed the observations with missing values). We then build a large regression tree on the training data and vary α in order to create subtrees with different number of leaves. Finally, we perform six-fold cross validation in order to estimate the cross-validated MSE (mean square error) of the trees as a function of α. The unpruned tree is showed in Figure 7.1.4. In Figure 7.1.5, we see the cross-validation MSE (we also see that the parameter α leading to a tree with 3 leaves is 0.077, 3 leaves seems to be the optimal size). The pruned tree (with 3 leaves) can be seen in Figure 7.1.6.

7.1.2. Classification trees. A classification tree is very similar to a regression tree, except that it is used to predict a qualitative response rather than a quantitative one. In a regression tree, the predicted response is a mean over the training outputs in a given box. For a classification tree, the predicted response is the most commonly occurring output in a given box. In interpreting the results, we will be often interested not only in the class prediction but also in the class proportion among the training outputs that fall into a given region.

When growing a classification tree, we use binary splitting. Instead of the RSS, we could use the classification error rate as a criterion for making the binary splits. This classification error rate is the fraction of the training outputs that do not belong to the most common class:

$$E = 1 - \max_k(\widehat{p}_{m,k}),$$

where $\widehat{p}_{m,k}$ represents the proportion of training outputs in the m-th region that are from the k-class. It turns out this error is not sensitive enough for tree-growing and in practice, we use to other measures.

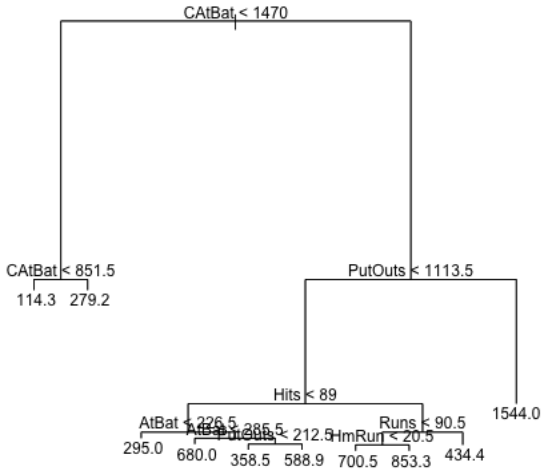


FIGURE 7.1.4. Regression tree analysis for the `Hitters` data. This unpruned tree is the result from top-down greedy splitting on the training data.

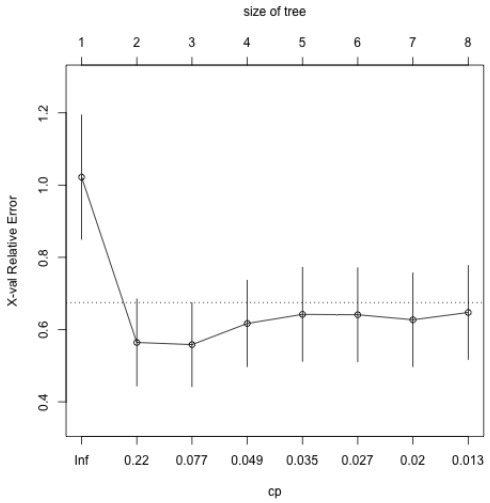


FIGURE 7.1.5. The cross-validation MSE.

The Gini index is defined by

$$G = \sum_{k=1}^{K} \widehat{p}_{m,k}(1 - \widehat{p}_{m,k}) .$$

This Gini index takes a small value if all the $\widehat{p}_{m,k}$'s are close to zero or one. For this reason, the Gini index is referred to as a measure of node purity (when it is small, it means that a node contains predominantly observations from a single class).

An alternative to the Gini index is the entropy, given by

$$D = -\sum_{k=1}^{K} \widehat{p}_{m,k} \log(\widehat{p}_{m,k}) .$$

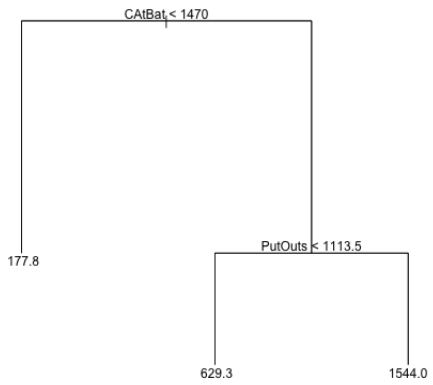


FIGURE 7.1.6. Pruned tree.

As $0 \leq \widehat{p}_{m,k} \leq 1$, we have $0 \leq -\widehat{p}_{m,k}\log(\widehat{p}_{m,k})$. One can show that the entropy will take on a value near 0 if the $\widehat{p}_{m,k}$'s are all near zero or near one. As a consequence, the entropy will take a small value if the m-th node is pure. Numerically, the Gini index and the entropy are quite similar.

When building a classification tree, the Gini index and the entropy are used to evaluate the quality of a particular split. Any of these approaches might be used when pruning the tree, but the classification error rate is preferable as prediction accuracy of the final pruned tree is the goal.

In Figure 7.1.7, we see the unpruned tree built on the `heart` dataset (`library(kmed)`). This data contains a qualitative outcome `class` for 297 patients who presented with chest pain (`class` goes from 0 ="no heart disease" to 4 ="very likely to have heart disease"). The outcome has been measured with an angiographic test. There are 13 features including `age`, `sex`, `chol` (for cholesterol), and other heart and lung functions measurements. Cross-validation results in a tree with four leaves (see Figure 7.1.8).

Thus far, we have assumed that the features were quantitative. But decision trees can be constructed even in the presence of qualitative features. For example, in the `heart` data, some of the features such as `sex`, `exang` (exercise induced angina) are qualitative. A split on one of these variables amounts to assigning some of the qualitative values to one branch and assigning the remainder to another branch. In Figure 7.1.7, the top node corresponds to splitting `thal`. The text `thal:a` indicates that the left-hand branch coming out of that node consists of observations with the first value of the `thal` variable (normal), and the right-hand node consists of the remaining observations (fixed or reversible defects).

We see in Figure 7.1.7 that some splits yield two leaves that have the same predicted value. The split is performed because it leads to increased node purity.

7.1.3. Comparison with linear models. Tree methods are very different from linear methods for regression and qualification. We have seen such methods in Chapter 2 (linear regression), Chapter 3 (logistic regression), Chapter 4 (SVM). A linear regression assumes a model of the form

$$f(X) = \beta_0 + \sum_{j=1}^{D} \beta_j X_j\,, \tag{7.3}$$

($f(X)$ is the output for the input $X = (X_1, \ldots, X_D)^T$) whereas regression trees assume a model of the form

$$f(X) = \sum_{m=1}^{M} c_m \mathbb{1}_{\{X \in R_m\}}\,, \tag{7.4}$$

where $R_1, \ldots, R_M$ represent a partition of the input space as in Figure 7.1.3.

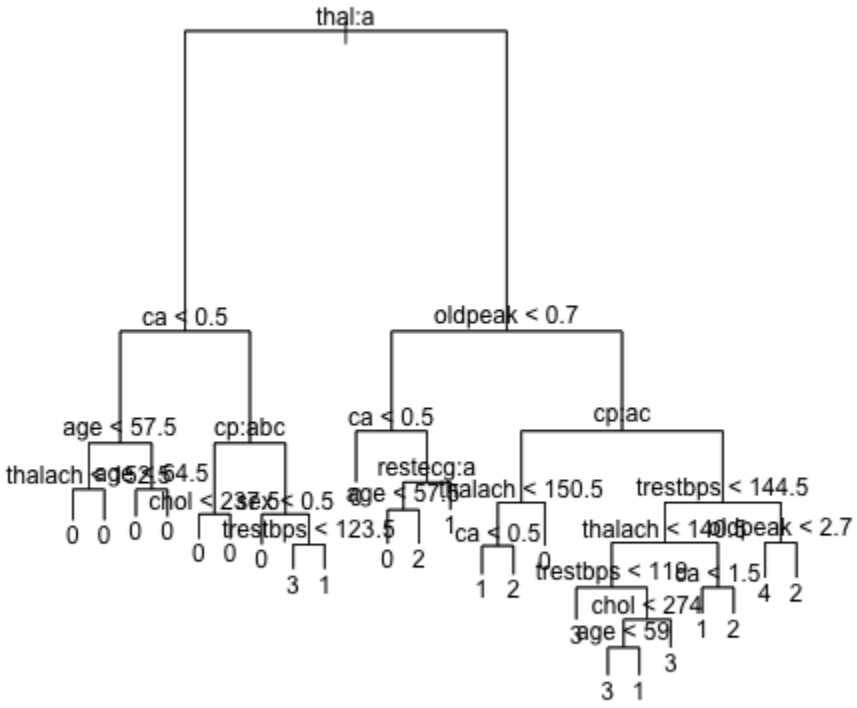


FIGURE 7.1.7. `heart` data, the unpruned tree

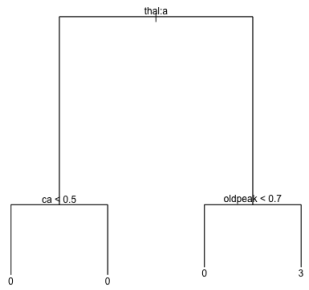


FIGURE 7.1.8. `heart` data, the pruned tree

If the relationship between the output and the input is well approximated by (7.3), then an approach such as linear regression will work well, and will outperform a method such as regression tree that does not exploit this linear structure. If there is a highly non-linear and complex relationship between the input and the output, as in Equation (7.4), then decision trees may outperform classical approaches. We can see an example in Figure 7.1.9. The relative performances of tree-based and classical

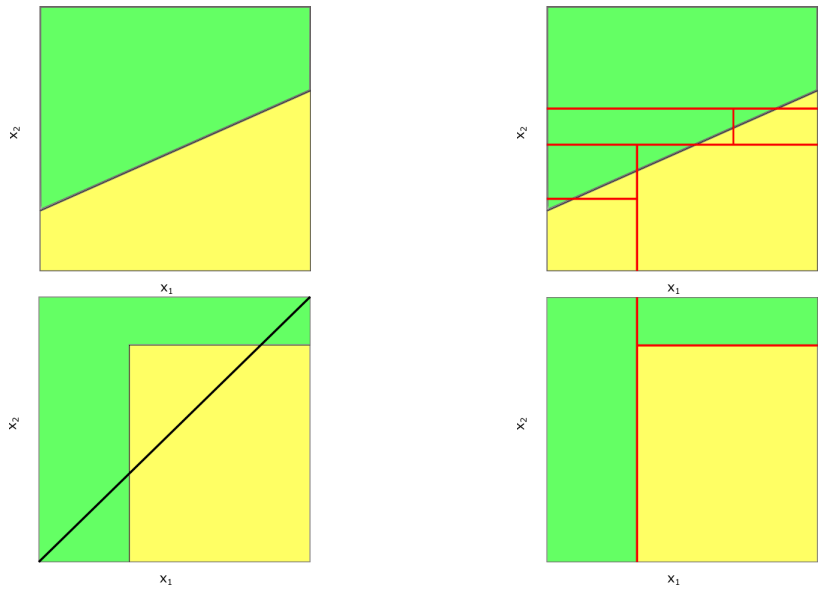


FIGURE 7.1.9. Top row: A two-dimensional classification example in which the true decision boundary is linear (the boundary between the yellow area and the green area. A classical approach that assumes a linear boundary (left) will perform better than a decision tree that performs splits parallel to the axes (right). Bottom row: Here the true decision boundary is non-linear and a linear model is unable to capture the true decision boundary (left) whereas a decision tree is successful (right).

approaches can be assessed by estimating the test error, using either cross-validation or the validation set approach.

7.1.4. Advantages/disadvantages of trees. Decision trees have a number of advantages over the more classical approaches of Chapter 2 (linear regression), Chapter 3 (logistic regression), Chapter 4 (SVM).

Advantages.

- Easy to explain (even easier than linear regression).
- It mirrors the human decision-making (more than regression and classification approach).
- Graphical display can be interpreted by a non-expert.
- They can handle qualitative features (no need to create dummy variables)

Disadvantages.

- They do not have the same level of predictive accuracy as some of the other regression and classification approach.
- Trees can be very non-robust, meaning that a small change in the data can cause a large change in the tree.

In the next Section, we will see how to overcome these problems using methods called bagging and random forests

7.2. Bagging, random forests

7.2.1. Bagging. The decision trees introduced in the previous Section suffer from high variance. This means that if we split the training data into two parts at random, and fit a decision tree to both halves, the results that we get can be quite different. A procedure with low variance will yield similar results if applied to different data sets; linear regression tends to have low variance if the ratio of N to D is moderately large (N is the numbers of points in our data set and D is the number of dimensions/features). Bootstrap aggregation or bagging is a procedure that allows to reduce the variance of a statistical

learning method. It is very useful and frequently used in the context of decision trees. (See an good introduction to bootstrap in [**GWHT17**], Section 5.2, p. 187.)

For a given set of N independent observations $Z_1, \ldots, Z_N$, each with variance σ^2, the variance of the mean $\overline{Z}$ is σ^2/N. So, averaging a set of observations reduces the variance. Hence, when we have a training set, we divide it into B folds, compute prediction functions $\widehat{f}^1, \widehat{f}^2, \ldots, \widehat{f}^B$ using each of the folds, and average them in order to obtain a single low-variance prediction function given by

$$x \mapsto \widehat{f}_{\text{avg}}(x) = \frac{1}{B} \sum_{b=1}^{B} \widehat{f}^b(x) .$$

This is usually not practical because we do not have access to multiple training sets. Instead, we can bootstrap: we take repeated samples from the (single) training data set. In this approach, we create B different bootstrapped training data sets (meaning we have sampled with replacement from the original data set to create each bootstrapped data set). We compute a prediction function on the b-th bootstrapped training set: we get $\widehat{f}^{*b}$, and finally average all the predictions, to obtain the following prediction function

$$x \mapsto \widehat{f}_{\text{bag}}(x) = \frac{1}{B} \sum_{b=1}^{B} \widehat{f}^{*b}(x) .$$

This is called bagging.

This bagging procedure is particularly useful for regression trees. To apply it to regression trees, we simply construct B regression trees using B bootstrapped training sets, and average the resulting predictions. In this case, the trees are grown deep (they are not pruned). Hence, each individual tree has high variance, but low bias. Averaging over the B trees reduces the variance. This method has been proved to give impressive improvements in accuracy by combining together hundreds or thousands of trees into a single procedure.

Thus far, we have described bagging in the regression context. We are going to extend it to a classification problem where the output is qualitative. For a given input, we record the class predicted by each of the B trees, and take a majority vote: the prediction is the most commonly occurring class amongst the B predictions.

Out-of-Bag error estimation. We can perform a straightforward estimation of the test error of a bagged model, without using cross-validation and validation test set. Recall that in bagging, trees are repeatedly fit to bootstrapped subsets of the observations. One can show that, on average, each bagged tree makes use of around two-thirds of the observations. The remaining one-third of the observations not used to fit the given bagged tree are called out-of-bag (OOB) observations. We can predict the output for the i-th input using each of the trees in which that input/observation was OOB. This will yield around $B/3$ predictions for the i-th input. In order to obtain a single prediction for the i-th input, we can average these predicted responses (if we are doing a regression) or take a majority vote (if we are doing a classification). This leads to a single OOB prediction for the i-th input. An OOB prediction can be obtained in this way for each of the N inputs, from which the overall OOB mean square error (for a regression problem) or classification error (for a classification problem) can be computed. The resulting OOB error is a valid estimate of the test error for the bagged predictor. since the response for each input is computed using only the trees that were not fit using that observation.

Variable importance measurements. Bagging typically results in improved accuracy over prediction using a single tree. Unfortunately, it can be difficult to interpret the result (one of the advantages of decision trees is the easily interpretable diagram that results from the procedure). Because when we have a large number of trees, it is no longer possible to represent our result using a single tree, and it is no longer clear which variables are most important to the procedure.

Nevertheless, we can obtain an overall summary of the importance of each feature using the RSS (residual sum of squares) (for regression) or the Gini index (for classification). In the case of bagging regression trees, we can record the total amount that the RSS is decreased due to splits over a single feature, averaged over all B trees. A large value indicates an important feature. Similarly, in the context of bagging classification trees, we can add up the total amount that the Gini index is decreased by splits over a given feature, averaged over all B trees.

7.2.2. Random forests. Random forests provide an improvement over bagged trees by way of a small tweak that decorrelates the trees. As in bagging, we build a number of decision trees on bootstrapped training samples. But when building these decision trees, each time a split in a tree is considered, a random sample of m features is chosen as split candidates from the full set of D features. The split is allowed to use only one of those m features. A fresh sample of m features is taken at each split, and typically, we choose $m \approx \sqrt{D}$.

In other words, in building a random forest, at each split in the tree, the algorithm is not even allowed to consider a majority of the available features. The rationale behind this procedure is the following. Suppose there is one very strong feature in the data set, along with a number of other moderately strong features (strong meaning they are important in making a good prediction). Then, in the collection of bagged trees, most of the trees will use this strong feature in the top split. So all the bagged trees will look quite similar to each other. Hence the prediction from the bagged trees will be highly correlated. And averaging many highly correlated quantities does not lead to a large reduction in variance.

Random forests overcome this problem by forcing each split to consider only a subset of the features. Therefore, on average $(D-m)/D$ of the splits will not even consider the strong features, and so other features will have more of a chance. We can think of this process as decorrelating the trees, thereby making the average of the resulting trees less variable and hence more reliable.

The main difference between bagging and random forests is the choice of the predictor subset size m. If a random forest is built using $m = D$, then this amounts simply to bagging.

7.3. Examples in R

7.3.1. Baseball (`Hitters` dataset). We use the `Hitters` data set (comes with `library(IRLS)`) to predict a baseball player's `Salary` given: `Years` (the number of years he has played in the major leagues) and `Hits` (the number of hits he made in the previous year). We remove observations with missing `Salary` and take the logarithm of `Salary` (`Salary` is measured in thousands of dollars).

Using the `rpart` command. We load some libraries.

```
library(ISLR)
library(rpart)
library(rpart.plot)
```

We draw a graph of `Years` vs `Salary` (see Figure 7.1.2).

```
plot(Hitters$Years, Hitters$Hits, xlab="Years", ylab="Hits")
```

We grow a regression tree:

```
tree <- rpart(Salary ~ Years + HmRun, data=Hitters, control=rpart.control(cp=.0001))
```

where

- `Salary ~ Years + HmRun` is the formula (we want to explain `Salary` by `Years` and `HmRun`)
- we specify the data with `data=Hitters`
- `method=''anova''` for regression, `''class''` for classification
- `control=...` are optional parameters for controlling tree growth. For example, `control=rpart.control(minsplit=30, cp=0.001)` requires that the minimum number of observations in a node be 30 before attempting a split and that a split must decrease the overall lack of fit by a factor of 0.001 (cost complexity factor) before being attempted.

We can have a look at the variable produced:

```
printcp(tree)
```

and draw the tree itself

```
prp(tree)
```

(see Figure 7.3.1). Let us have a look at the `cp` table (`CP` is our parameter α of Equation (7.2))

```
>tree$cptable
```

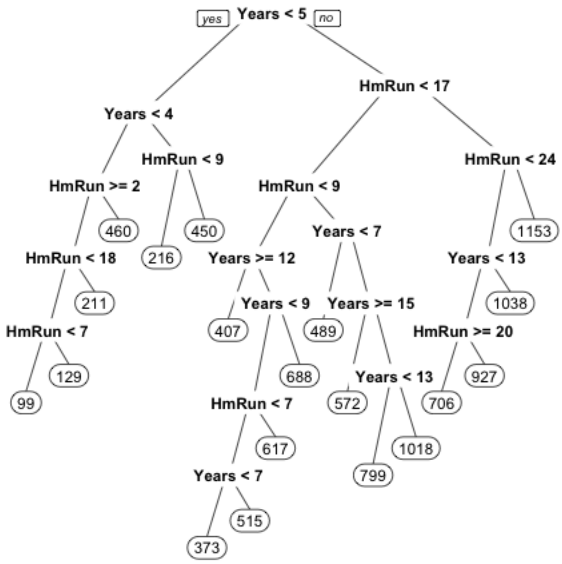


FIGURE 7.3.1. Full regression tree for `Hitters`

```
      CP          nsplit  rel error    xerror       xstd
1   0.2467499648       0 1.0000000 1.0047200 0.1377826
2   0.1080693184       1 0.7532500 0.7590044 0.1268194
3   0.0186561017       2 0.6451807 0.7566213 0.1301316
...      ...             ...         ...         ...
```

We have various α's in the column `CP`, the RSS error in the column `xerror`. We can plot `xerror` vs `CP`:

```
plotcp(tree)
```

(which gives something similar to Figure 7.1.5). We can find what is the best parameter and then prune with this parameter:

```
best <- tree$cptable[which.min(tree$cptable[,"xerror"]),"CP"]
pruned_tree <- prune(tree, cp=best)
```

We plot the pruned tree:

```
prp(pruned_tree,
        faclen=0, #use full names for factor labels
        extra=1, #display number of obs. for each terminal node
        roundint=F, #don't round to integers in output
        digits=5) #display 5 decimal places in output
```

(see Figure 7.3.2) To get a nicer plot, we can use:

```
rpart.plot(prund_tree)
```

Prediction for new data:

```
predict(pruned_tree,newdata=....)
```

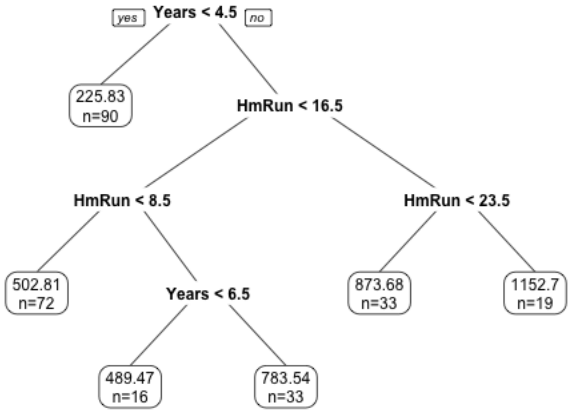


FIGURE 7.3.2. Pruned regression tree for `Hitters`

Using the `tree` command. Alternatively, we could use the `tree` command. First, we remove rows with missing values (`tree` cannot deal with missing values, on the contrary to `rpart`):

```
Hitters_clean <- NULL
for (i in 1:nrow(Hitters))
{
        if (!(any(is.na(Hitters[i,,]))))
        {
                Hitters_clean <- rbind(Hitters_clean,Hitters[i,])
        }
}
```

(here the `!(condition)` means: NOT `(condition)`). We extract a training set:

```
nrow(Hitters_clean)
trainHit <- sample(1:nrow(Hitters_clean),132)
```

We build a tree aiming at explaining `Salary` with all the other variables:

```
library(tree)
treeHit <- tree(Salary~ .,data=Hitters_clean,subset=trainHit)
plot(treeHit)
text(treeHit,pretty=0)
```

(see resulting graph in Figure 7.1.1).

We perform a cross-validation with six folds to find the best parameter α:

```
cvTree <- cv.tree(treeHit,K=6)
```

We can plot `cvTree$size` vs `cvTree$dev` (the deviance, here the MSE)

```
plot(cvTree$size[1:(l-1)],cvTree$dev[1:(l-1)],type='b',
        xlab='Tree size',ylab='Mean square error')
```

(result may vary as the optimization procedure rely on some randomness). After this, we have to choose which is the best tree size (around 3) and prune:

```
prune.hitters <- prune.tree(treeHit,best=3)
```

(`best` is the size of the tree after pruning).

Make prediction for new data

```
predict(pruneHit,newdata=Hitters_clean,subset=-trainHit)
```

(here we want predictions for the data *not* in the training set)

7.3.2. Boston Housing dataset (same as in Chapter 2). The data is included in the `MASS` library.

```
library(MASS)
Boston
```

Regression tree. We create a training set and fit the tree to the training data.

```
set.seed(234)
train = sample(1:nrow(Boston), nrow(Boston)/2)
treeBoston=tree(medv ~ .,Boston,subset=train)
summary(treeBoston)
```

We get

```
Regression tree:
tree(formula = medv ~ ., data = Boston, subset = train)
Variables actually used in tree construction:
[1] "lstat"   "rm"      "ptratio" "crim"    "nox"
Number of terminal nodes:  10
Residual mean deviance:  14 = 3403 / 243
Distribution of residuals:
Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-16.3000  -2.1500  -0.4027   0.0000   1.6970  28.2500
```

We see that only three variables have been used in constructing the tree. In the context of regression, the deviance is simply the sum of squared errors for the tree. We now plot the tree

```
plot(treeBoston)
text(treeBoston,pretty=0)
```

(see the result in Figure 7.3.3) The variable `lstat` measures the percentage of individuals with lower socioeconomic status. The tree indicates that lower values of `lstat` correspond to more expensive houses. The tree predicts a median house price of $46, 400 for larger homes in suburbs in which residents have high socioeconomic status (`rm>=7.437` and `lstat<9.715`).

Now we use the `cv.tree()` function to see whether pruning the tree will improve performance.

```
cvBoston <- cv.tree(treeBoston,K=6)
plot(cvBoston$size,cvBoston$dev,type='b')
```

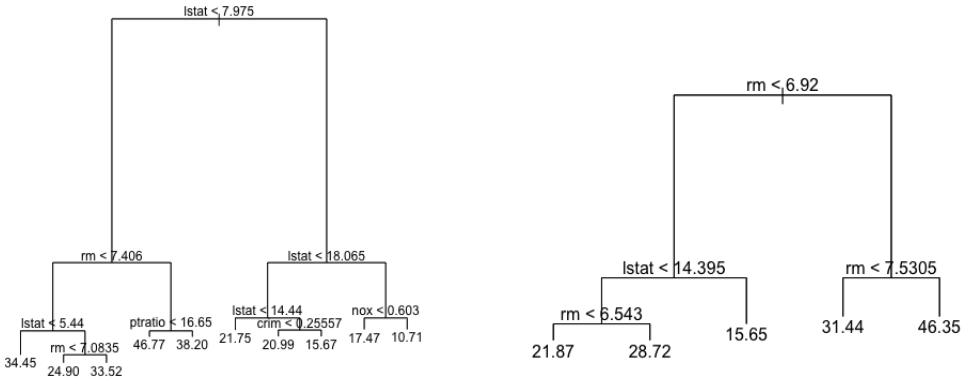
In this case, the most complex tree is selected by cross-validation. However, if we wish to prune the tree, we could do so as follows, using the `prune.tree()` function:

```
pruneBoston=prune.tree(treeBoston,best=5)
plot(pruneBoston)
text(pruneBoston,pretty=0)
```

(see the result in Figure 7.3.3).

In keeping with the cross-validation results, we use the unpruned tree to make predictions on the test set.

```
yhat=predict(treeBoston,newdata=Boston[-train,])
bostonTest=Boston[-train,"medv"]
plot(yhat,bostonTest)
abline(0,1)
```



(A) Boston housing full tree (B) Boston housing pruned tree

FIGURE 7.3.3. Boston housing trees

(see the result in Figure 7.3.4). In Figure 7.3.4, the points are grouped around the line "$y = x$" so our predictions are good. We have

```
> mean((yhat-bostonTest)^2)

26.31314
```

so the test MSE associated with the regression tree is 26.31314. The square-root of this MSE is around 5.130, indicating that this model leads to test predictions that are within around \$5,130 of the true median home value for the suburb.

Bagging and random forests. Now, we want to apply bagging and random forests to the `Boston` data. We perform bagging as follows:

```
library(randomForest)
set.seed(1)
bagBoston=randomForest(medv~.,data=Boston,subset=train,
mtry=13,importance=TRUE)
```

where

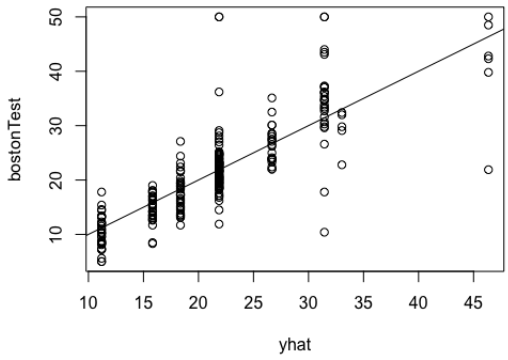
- `mtry=13` means that all 13 features should be considered for each split (so we perform bagging),
- `importance=TRUE` means we assess the importance of features.

We get

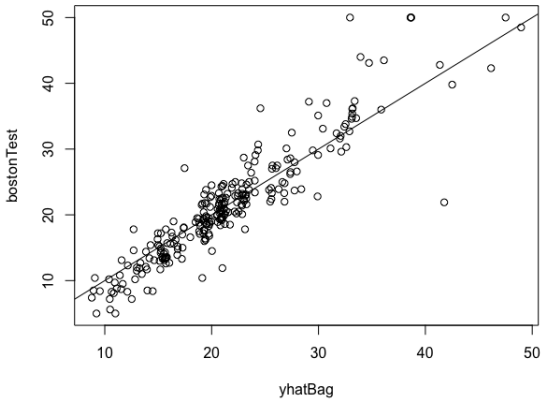
```
> bagBoston

Call: randomForest(formula = medv ~ ., data = Boston, mtry = 13, importance = TRUE,
subset = train)                Type of random forest: regression
Number of trees: 500
No. of variables tried at each split: 13
          Mean of squared residuals: 15.19213
% Var explained: 82.98
```

How well does this bagged model perform on the test set?



(A) For a single tree



(B) For a bagged tree

FIGURE 7.3.4. Predictions vs reality for `Boston` (on the test set)

```
 yhatBag = predict(bagBoston,newdata=Boston[-train,])
plot(yhatBag, bostonTest)
abline(0,1)
```

(see the result in Figure 7.3.4). The points are again grouped around the line "y=x" so our predictions are good. We have

```
>    mean((yhatBag-bostonTest)^2)

12.40002
```

around a half of what we got for the single tree regression. We could change the number of trees grown by `randomForest` using the `ntree` argument

```
bagBoston=randomForest(medv~.,data=Boston,subset=train, mtry=13,ntree=25)
yhatBag = predict(bagBoston,newdata=Boston[-train,])
mean((yhatBag-bostonTest)^2)
```

We get a MSE of

```
13.43888
```

(so no big change here).

Growing a random forest proceeds in exactly the same way, except that we use a smaller value of the `mtry` argument. By default, `randomForest()` uses $D/3$ variables when building a random forest of regression trees, and D variables when building a random forest of classification trees. Here we use `mtry = 6`.

```
rfBoston=randomForest(medv~.,data=Boston,subset=train,
                                                    mtry=6,importance=TRUE)
yhat.rf = predict(rfBoston,newdata=Boston[-train,])
mean((yhat.rf-bostonTest)^2)
```

We get a MSE of

```
11.65072
```

This is an improvement over the bagging method.

Using the `importance()` function, we can view the importance of each variable.

```
importance(rfBoston)
```

	%IncMSE	IncNodePurity
crim	8.881849	745.42211
zn	2.034618	62.86110
indus	9.568684	957.70779
chas	2.138816	193.47944
nox	10.697316	805.75171
rm	34.524482	8391.58450
age	11.190707	557.15096
dis	9.702571	1143.41290
rad	3.508583	97.44346
tax	6.797353	370.45004
ptratio	11.128163	1107.02730
black	6.191073	418.99133
lstat	29.259320	7280.24124

Two measures of variable importance are reported. The former is based upon the mean decrease of accuracy in predictions on the out of bag samples when a given variable is excluded from the model. The latter is a measure of the total decrease in node impurity that results from splits over that variable, averaged over all trees. In the case of regression trees, the node impurity is measured by the training RSS, and for classification trees by the deviance. Plots of these importance measures can be produced using the `varImpPlot()` function:

```
varImpPlot(rfBoston)
```

(see the result in Figure 7.3.5).

7.3.3. Titanic dataset. The data has 1046 observations on 6 variables (and comes with `library(rpart.plot)`)

- `pclass`: passenger class
- `survived`: died or survived
- `sex`: male or female
- `age`: age in years
- `sibsp`: number of siblings or spouses aboard
- `parch`: number of parents or children aboard

We load the libraries we need.

```
library(tree)
library(rpart.plot)
```

We create a training set

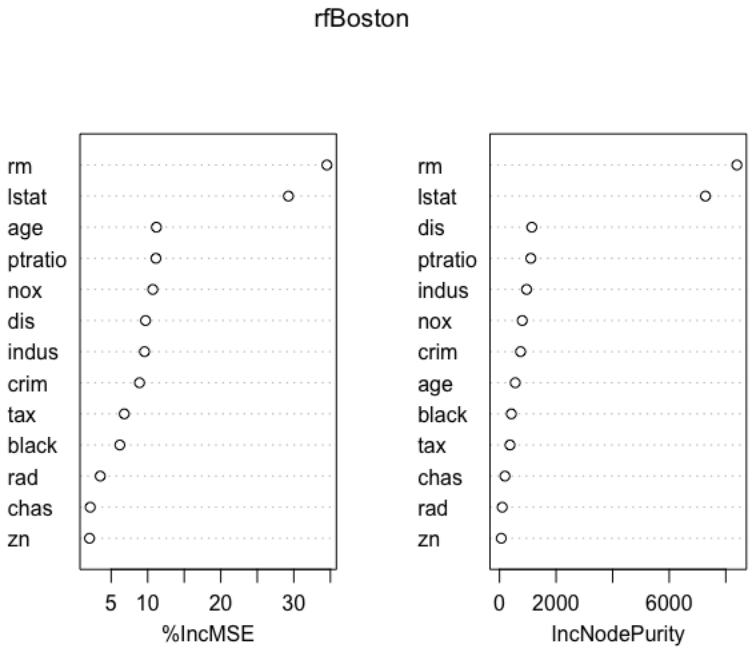


FIGURE 7.3.5. Importance of the variables

```
train <- sample(1:nrow(ptitanic), 1000)
```

and a tree (based on the training set)

```
treeTi <- tree(survived ~ .,data=ptitanic,subset=train,
control=tree.control(nobs=1000,mindev=0.005))
```

where

- `nobs` is the number of observations (1000 in the training set)
- `mindev` is such that the within-node deviance must be at least this times that of the root node for the node to be split (in a classification context, deviance = number of misclassifications)

We plot the tree:

```
plot(treeTi)
text(treeTi,pretty=0)
```

We perform cross-validation with 10 folds

```
cvTi <- cv.tree(treeTi,K=10)
plot(cvTi$size,cvTi$dev,type='b')
```

and we see that the deviance is minimal for a size around 6.

We prune the tree (until its size is 6) and plot the pruned tree

```
pruneTi <- prune.tree(treeTi,best=6)
plot(pruneTi)
text(pruneTi)
```

We make predictions with the pruned tree:

```
yhat <- as.data.frame(predict(pruneTi,newdata=ptitanic[-train,]))
```

We get probabilities (of survival) which we convert into factors ('died' or 'survived'):

```
yhat$survived <- ifelse(yhat$died > 0.5,'died','survived')
```

We build the confusion matrix

```
treeTest=ptitanic[-train,'survived']
cm <- table(yhat$survived,treeTest)
library(caret)
confusionMatrix(cm)
```

Before using `randomForest`, we must clean the data of the `NA` entries (`tree` can deal with it but not `randomForest`):

```
library(randomForest)
titanic <- NULL
for (i in 1:nrow(ptitanic))
{
        if (!(any(is.na(ptitanic[i,]))))
        {
                titanic <- rbind(titanic,ptitanic[i,])
        }
}
```

We build a training set

```
nrow(titanic)
train <- sample(1:nrow(titanic), 800)
```

We perform bagging

```
bagTi <- randomForest(survived ~ ., data=titanic,subset=train,mtry=5,importance=TRUE)
bagTi
```

We make predictions with this bagged forest:

```
yBag <- predict(bagTi,newdata=titanic[-train,])
```

And compare our predictions to the data:

```
treeTest <- titanic[-train,'survived']
cm <- table(yBag,treeTest)
confusionMatrix(cm)
```

(no real change in accuracy when compared to the pruned tree predictions).

We than try a random forest approach:

```
rfTi <- randomForest(survived ~ ., data=titanic,subset=train,mtry=3,importance=TRUE)
yRF <- predict(rfTi,newdata=titanic[-train,])
cm <- table(yRF,treeTest)
confusionMatrix(cm)
```

(no gain in accuracy).

We check the importance of the variables:

```
importance(rfTi)
varImpPlot(rfTi)
```

7.3.4. In-vehicle coupon recommendation data set ([WRDV+17]). This data[1] was collected via a survey on Amazon Mechanical Turk. The survey describes different driving scenarios including the destination, current time, weather, passenger, etc., and then ask the person whether he will accept the coupon if he is the driver. The data set is `in-vehicle-coupon-recommendation.csv`.

(1) Load the data into R and convert all characters (chr) into factors (a tedious task).
(2) Split your data into a training set and a test set.

[1] `https://archive.ics.uci.edu/ml/datasets/in-vehicle+coupon+recommendation`

(3) Fit a decision tree on the training set (show the tree).
(4) Test your tree on the test set (show the confusion matrix and the accuracy).
(5) Use cross-validation to find the best pruning of this tree (the pruned tree should be around 5 leaves). Show the pruned tree.
(6) Test your pruned tree as above.
(7) Use bagging to improve the accuracy.
(8) Try random forests to improve the accuracy. .

7.3.5. Useful commands in R. If `A` is a boolean (i.e. takes values `TRUE` or `FALSE`) then `!A` means NOT(`A`).

Using `library(rpart)`. Grow a tree:

```
library(rpart)
tree <- rpart(Salary~Years+HmRun, data=Hitters, control=rpart.control(cp=.0001))
```

where

- `Salary ~ Years + HmRun` is the formula (we want to explain `Salary` by `Years` and `HmRun`)
- we specify the data with `data=Hitters`
- `method=''anova''` for regression, `''class''` for classification
- `control=...` are optional parameters for controlling tree growth. For example, `control=rpart.control(minsplit=30, cp=0.001)` requires that the minimum number of observations in a node be 30 before attempting a split and that a split must decrease the overall lack of fit by a factor of 0.001 (cost complexity factor) before being attempted.

Summary:

```
printcp(tree)
```

error as a function of `CP`:

```
plotcp(tree)
```

plot the tree:

```
prp(tree)
```

or

```
rpart.plot(tree)
```

prune the tree:

```
pruned_tree <- prune(tree, cp=best)
```

(`cp` is the parameter α).

Prediction for new data

```
predict(pruned_tree, newdata=...)
```

Using `library(tree)`. Building a tree (using only a subset of the data)

```
library(tree)
treeHit <- tree(Salary~ ., data=Hitters_clean, subset=trainHit)
```

One can add the option `control=tree.control(nobs=1000,mindev=0.005)` where

- `nobs` is the number of observations (1000 in the training set)
- `mindev` is such that the within-node deviance must be at least this times that of the root node for the node to be split (in a classification context, deviance = number of misclassifications)

To plot the tree with text, we need the two following lines

```
plot(treeHit)
plot(treeHit, type='uniform') #to get branches of uniform length
text(treeHit, pretty=0)
```

(the `pretty` option can be used to tweak a little bit the graphical representation).

Cross-validation to find the best α:

```
cvTree <- cv.tree(treeHit,K=6)
```

Prune a tree, specifying what size should be the resulting tree:

```
pruneHit <- prune(treeHit,best=3)
```

(here we want a pruned tree of size 3).

Prediction for new data

```
predict(pruneHit,newdata=...)
```

Using `library(randomForest)` for bagging and random forests. Perform bagging:

```
library(randomForest)
bagBoston=randomForest(medv~.,data=Boston,subset=train,
mtry=13,ntree=25,importance=TRUE)
```

where

- `mtry=13` means that all 13 features should be considered for each split (so we perform bagging),
- `importance=TRUE` means we assess the importance of features,
- `ntree` is the number of tree (if not specified, `R` will choose it by itself).

To build a random forest, we just need to choose a lower `mtry`.

To see importance of each variable:

```
importance(bagBoston)
```

and to plot importance of each variable:

```
varImpPlot(bagBoston)
```

CHAPTER 8

Markov chain Monte-Carlo inference

A recent survey places the Metropolis algorithm among the ten algorithms that have had the greatest influence on the development and practice of science and engineering in the 20th century ([**BS00**]). (The Metropolis algorithm is a particular case of the MCMC methods.)

8.1. Introduction with an example

We are interested in the New-York stock exchange daily return (see Figure 8.1.1). We suppose this

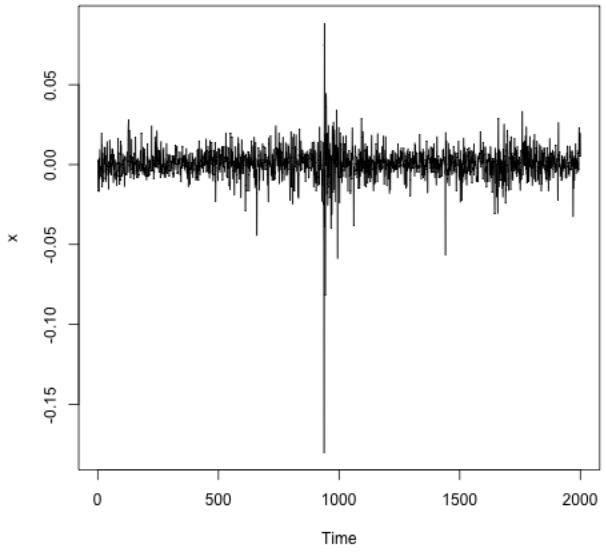


FIGURE 8.1.1. NYSE

sequence of numbers $(X_t)_{1\leq t\leq n}$ follows a GARCH(1,1) model, meaning

$$\begin{cases} X_t = \sigma_t e_t \text{ with i.i.d. } e_t \sim \mathcal{N}(0,1)\,, \\ \sigma_t = \alpha_0 + \alpha_1 X_{t-1} + \beta_1 \sigma_{t-1}^2\,. \end{cases}$$

We want to estimate α_0, α_1, β_1. We proceed with a Bayesian analysis of the problem. We have an a priori idea of the distribution of $(\alpha_0, \alpha_1, \beta_1)$ in the form of a density (the a priori density, also called prior)

$$p(\alpha_0, \alpha_1, \beta_1) = p(\alpha_0)p(\alpha_1)p(\beta_1)\,.$$

We have here a product of densities, meaning the three variables are independent.

⚠ In a true Bayesian fashion, we use the same letters for the variables and the densities argument. This might be misleading but also simplifies considerably the notations.

We are interested in the posterior density (also called posterior):

$$p(\alpha_0, \alpha_1, \beta_1 | x)\,,$$

where $x = (x_t)_{1\le t\le n}$ is the observation of the New-York stock exchange daily return we have . We will derive this posterior as a marginal of

$$p(\alpha_0, \alpha_1, \beta_1, \sigma_1 | x),$$

where $\sigma = (\sigma_t)_{1\le t\le n}$. We use the Bayes formula

$$p(\alpha_0, \alpha_1, \beta_1, \sigma_1 | x) = \frac{p(x|\alpha_0, \alpha_1, \beta_1, \sigma_1) p(\alpha_0, \alpha_1, \beta_1, \sigma_1)}{p(x)}.$$

The numerator of the above fraction is known, but the denominator is not. We simply write ($\propto$ stands for "proportional to")

$$p(\alpha_0, \alpha_1, \beta_1, \sigma_1 | x) \propto p(x|\alpha_0, \alpha_1, \beta_1, \sigma_1) p(\alpha_0, \alpha_1, \beta_1, \sigma_1).$$

The whole point of this Chapter is to show how to sample from this posterior, even if it is known only up to a multiplicative factor.

8.2. Gibbs sampler

8.2.1. Theoretical description. We want to compute a posterior of the form

$$p(\theta|x) \propto p(x|\theta) p(\theta).$$

The parameter θ we want to estimate lives in a high-dimensional space, say $\mathbb{R}^D$. The goal here is to build a sequence $(\theta^{(s)})_{s\ge 0}$ behaving similarly to a sample of $p(\theta|x)$.

We sample each feature/coordinate in turn, conditioned on the values of the other features. Suppose you have a vector $\theta^{(s)} = (\theta_1^{(s)}, \dots, \theta_D^{(s)})^T$. We generate a new sample (/a new point) $\theta^{(s+1)}$ by sampling each component in turn, based on the most recent values of the other variables. For $D = 3$, we have

$$\left\{ \begin{array}{lcl} \theta_1^{(s+1)} & \sim & p(\theta_1 | \theta_2^{(s)}, \theta_3^{(s)}) \\ \theta_2^{(s+1)} & \sim & p(\theta_2 | \theta_1^{(s+1)}, \theta_3^{(s)}) \\ \theta_3^{(s+1)} & \sim & p(\theta_3 | \theta_1^{(s+1)}, \theta_2^{(s+1)}). \end{array} \right. \tag{8.1}$$

What will happen is that the chain $\theta^{(s)}$ will have asymptotically the law $p(\dots|x)$. This chain is called the Gibbs sample. What is to be understood is that p is a law on $\mathbb{R}^3$ (let us call $Z = (Z_1, Z_2, Z_3)$ a variable of law p) and $p(\theta_1|\theta_2^{(s)}, \theta_3^{(s)})$ is the law $\mathscr{L}(Z_1 | Z_2 = \theta_2^{(s)}, Z_3 = \theta_3^{(s)})$. Of course, the scheme is implementable only if we can sample from this conditional probability and those in Equation (8.1).

In practice: start somewhere, discard some of the initial samples until the Markov chain has burned in, or entered its stationary regime.

8.2.2. Ising model. This model is very popular in statistical physics. Let $N \in \mathbb{N}$. Let

$$\Lambda = \{-N, -N+1, \dots, -1, 0, 1, \dots, N\}^2 \subset \mathbb{Z}^2$$

(we could replace the dimension 2 by $d \in \mathbb{N}^*$). We define the configuration space by

$$E = \{-1; 1\}^\Lambda$$

(we have applications $f : \Lambda \to \{-1; 1\}$). The cardinality of E is 2^{2N+1}. For $x \in E$, we set

$$H(x) = \frac{1}{2} \sum_{\substack{m,m' \in \Lambda \\ |m-m'|=1}} |x(m) - x(m')|^2,$$

where we sum over all the pairs m, m' of E which are neighbors. Observe that $H(x)$ is zero if x is constant. Let $\beta > 0$, we define a probability on E :

$$\pi(x) = \frac{1}{Z(\beta)} e^{-\beta H(x)},$$

with $Z(\beta) = \sum_{x\in E} e^{-\beta H(x)}$. The constant $Z(\beta)$ is not known.

We would like to sample from π. We partition Λ into

$$\Lambda^+ = \{(m_1, m_2) \in \Lambda; m_1 + m_2 \text{ even}\},$$
$$\Lambda^- = \{(m_1, m_2) \in \Lambda; m_1 + m_2 \text{ odd}\}.$$

If m belongs Λ^+ then all its neighbors are in Λ^- (the inverse is also true).

For $x \in E$, we set

$$x^+ = (x(m), m \in \Lambda^+),$$
$$x- = (x(m), m \in \Lambda^-).$$

The components x^+ et x^- are restrictions of x (respectively to Λ^+ et Λ^-) and we write $x = (x^+, x^-)$. We now want to compute $\pi(x^+|x^-)$ ($\pi(x^+|x^-) = \mathbb{P}(X^+ = x^+|X^- = x^-)$ where X is a random variable of law π). We have:

$$\begin{aligned}
\pi(x^+|x^-) &= \frac{\pi(x^+, x^-)}{\pi(x-)} \\
&= \frac{\pi(x^+, x^-)}{\sum_{\substack{y \in E \\ y^- = x^-}} \pi(y)} \\
&= \frac{\exp\left(-\beta \sum_{m\in\Lambda^+} \sum_{\substack{m'\in\Lambda^- \\ |m-m'|=1}} (x^+(m) - x^-(m'))^2\right)}{\sum_{\substack{y \in E \\ y^- = x^-}} \exp\left(-\beta \sum_{m\in\Lambda^+} \sum_{\substack{m'\in\Lambda^- \\ |m-m'|=1}} (y^+(m) - y^-(m'))^2\right)} \\
&= \frac{\exp\left(-\beta \sum_{m\in\Lambda^+} \sum_{\substack{m'\in\Lambda^- \\ |m-m'|=1}} (1+1-2x^+(m)x^-(m))\right)}{\sum_{\substack{y \in E \\ y^- = x^-}} \exp\left(-\beta \sum_{m\in\Lambda^+} \sum_{\substack{m'\in\Lambda^- \\ |m-m'|=1}} (1+1-2y^+(m)x^-(m'))\right)} \\
&= \frac{\exp\left(2\beta \sum_{m\in\Lambda^+} \sum_{\substack{m'\in\Lambda^- \\ |m-m'|=1}} x^+(m)x^-(m')\right)}{\sum_{\substack{y \in E \\ y^- = x^-}} \exp\left(2\beta \sum_{m\in\Lambda^+} \sum_{\substack{m'\in\Lambda^- \\ |m-m'|=1}} y^+(m)x^-(m')\right)} \\
&= \frac{\prod_{m\in\Lambda^+} \exp\left(2\beta x^+(m) \sum_{\substack{m'\in\Lambda^- \\ |m-m'|=1}} x(m')\right)}{\sum_{\substack{y \in E \\ y^- = x^-}} \exp\left(2\beta \sum_{m\in\Lambda^+} \sum_{\substack{m'\in\Lambda^- \\ |m-m'|=1}} y^+(m)x^-(m')\right)}.
\end{aligned}$$

We have here a product of functions $x^+(m)$ over $m \in \Lambda^+$. So, under the law $\pi(.|x^-)$, the components $X^+(m)$, $m \in \Lambda$ are independent and of law

$$\mathbb{P}(X^+(m) = x^+(m)|X^- = x^-) \propto \exp\left(2\beta x^+(m) \sum_{\substack{m'\in\Lambda^- \\ |m-m'|=1}} x^-(m')\right).$$

So, with $M = \sum_{\substack{m'\in\Lambda^- \\ |m-m'|=1}} x^-(m')$

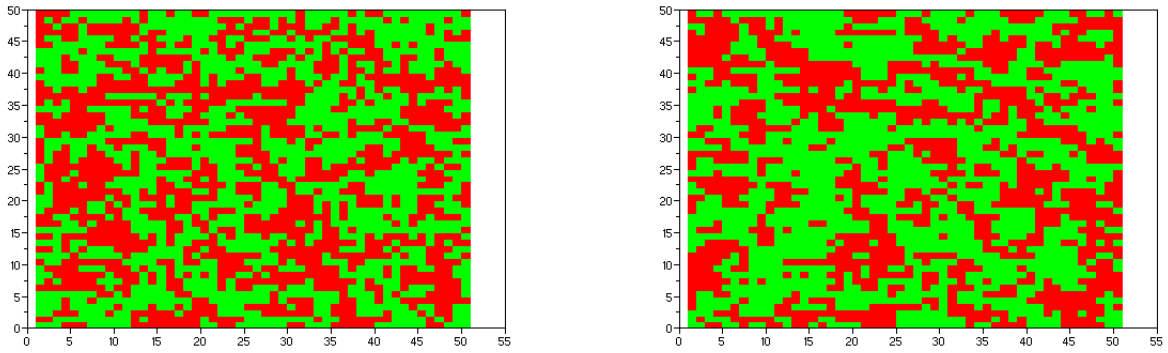
$$\mathbb{P}(X^+(m) = 1|X^- = x^-) = \frac{e^{2\beta M}}{e^{2\beta M} + e^{-2\beta M}},$$
$$\mathbb{P}(X^+(m) = -1|X^- = x^-) = \frac{e^{-2\beta M}}{e^{2\beta M} + e^{-2\beta M}}.$$

Then it is easy to sample form $\pi(.|x^-)$. For the same reason, it is easy to sample from $\pi(.|x^+)$. Let $(X_n)_{n\geq 0}$ be a sequence obtained by the Gibbs sampler algorithm (whose recurrence is described in Equation

(8.1):

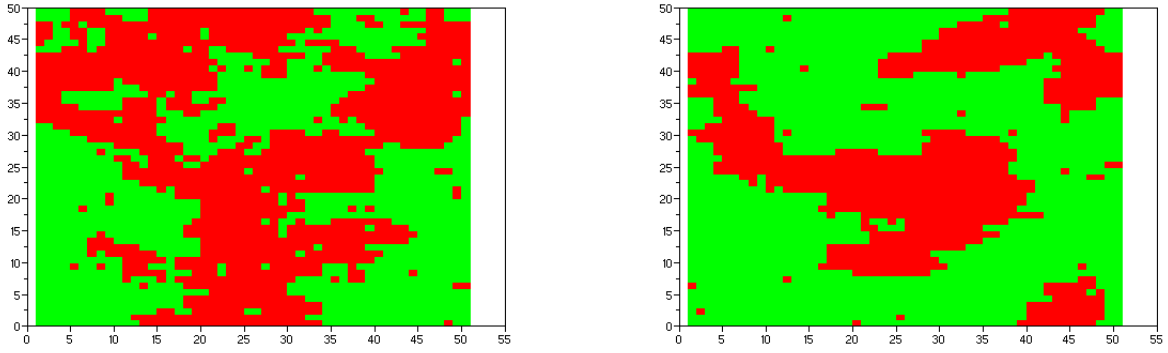
$$\begin{array}{c} X_1^+ \sim \pi(\ldots|X_0^-),\ X_1^- = X_0^+ \\ X_2^- \sim \pi(\ldots|X_1^+),\ X_2^+ = X_1^- \\ X_3^+ \sim \pi(\ldots|X_2^-),\ X_3^- = X_2^- \\ \ldots \end{array}$$

For n going to infinity, the law of X_n is close to the law π.



(A) $\beta = 0,3$ (B) $\beta = 0,5$

FIGURE 8.2.1



(A) $\beta = 0,8$ (B) $\beta = 1$

FIGURE 8.2.2

In Figures (8.2.1) and (8.2.2), we have sampled from law π using a Gibbs sampler (the results are thus *approximatively* of law π) for $N = 25$. The red dots (or dark grey) are $+1$ and the green dots (or light grey) are -1. We believe these realizations are representative of π. We observe that for a big β, the law π chooses preferably configurations x such that the number of neighbors with different values is small (we say that $m, m' \in \Lambda$ have different values if $x(m) \neq x(m')$). In our sample, the two colors are more mixed for a small β than for a big β.

8.2.3. Bayesian image analysis. In the Ising model, we associate to x of E a picture: $m \in \Lambda$ is a pixel, it is black if $x(m) = +1$ and white if $x(m) = -1$. We observe the pixels's colors and the observation gives us the exact color of each picture with probability p ($p \in]0;1[$). Then the a posteriori law, that is the law of $X = x$ knowing we observed y (the image is represented by a random variable X because we do not know it) is

$$\pi(x|y) \propto e^{-\beta H(x)} p^{a(x,y)} (1-p)^{d(x,y)},$$

where $a(x,y) = \#\{m \in \Lambda : x(m) = y(m)\}$, $d(x,y) = \#\{m \in \Lambda : x(m) \neq y(m)\}$. We want to sample from $\pi(.|y)$ (to get an idea of what was the original picture).

REMARK 8.1. The parameter p represents the observation error. For a small p, the observed image contains a lot of noise. The parameter β reflects our perception of the contrast in the original image. For a big β, we are looking for images with a high contrast.

We build a Gibbs sampler similar to what we have seen in the previous Section. The observation y is fixed and we want to sample from $\mu = (\mu(x), x \in E)$ with $\mu(x) = \pi(x|y)$ for all x. From the previous Section, we get:

$$\begin{aligned}
\mu(x^+|x^-) &= \frac{\mu(x^+, x^-)}{\sum_{\substack{z \in E \\ z^- = x^-}} \mu(z)} \\
&= \frac{\exp\left(-\beta \sum_{m \in \Lambda^+} \sum_{\substack{m' \in \Lambda^- \\ |m-m'|=1}} (x^+(m) - x^-(m'))^2\right) p^{a(x,y)} (1-p)^{d(x,y)}}{\sum_{\substack{z \in E \\ z^- = x^-}} \exp\left(-\beta \sum_{m \in \Lambda^+} \sum_{\substack{m' \in \Lambda^- \\ |m-m'|=1}} (z^+(m) - z^-(m'))^2\right) p^{a(z,y)} (1-p)^{d(z,y)}} \\
&= \frac{\prod_{m \in \Lambda^+} \exp\left(2\beta x^+(m) \sum_{\substack{m' \in \Lambda^- \\ |m-m'|=1}} x(m')\right)}{\sum_{\substack{y \in E \\ y^- = x^-}} \exp\left(2\beta \sum_{m \in \Lambda^+} \sum_{\substack{m' \in \Lambda^- \\ |m-m'|=1}} y^+(m) x^-(m')\right) p^{a(z,y)} (1-p)^{d(z,y)}} \\
&\qquad \times \prod_{m \in \Lambda^+} p^{1-|x^+(m)-y^+(m)|/2} (1-p)^{|x^+(m)-y^+(m)|/2} \\
&\qquad \times \prod_{m \in \Lambda^-} p^{1-|x^-(m)-y^-(m)|/2} (1-p)^{|x^-(m)-y^-(m)|/2} \\
&\propto \prod_{m \in \Lambda^+} \exp\left(2\beta x^+(m) \sum_{\substack{m' \in \Lambda^- \\ |m-m'|=1}} x(m')\right) p^{1-|x^+(m)-y^+(m)|/2} (1-p)^{|x^+(m)-y^+(m)|/2}.
\end{aligned}$$

We have here a product of functions of $x^+(m)$ over $m \in \Lambda^+$. So, under the law $\mu(.|x^-)$, the components $X^+(m)$, $m \in \Lambda$ are independent and of law

$$\mu(x^+(m)|x^-) \propto \exp\left(2\beta x^+(m) \sum_{\substack{m' \in \Lambda^- \\ |m-m'|=1}} x(m')\right) p^{1-|x^+(m)-y^+(m)|/2} (1-p)^{|x^+(m)-y^+(m)|/2}.$$

So, with $M = \sum_{\substack{m' \in \Lambda^- \\ |m-m'|=1}} x^-(m')$, if $y^+(m) = 1$,

$$\mu(x^+(m) = 1|x^-) = \frac{e^{2\beta M} p}{e^{2\beta M} p + e^{-2\beta M}(1-p)},$$

$$\mu(x^+(m) = -1|x^-) = \frac{e^{-2\beta M}(1-p)}{e^{2\beta M} p + e^{-2\beta M}(1-p)};$$

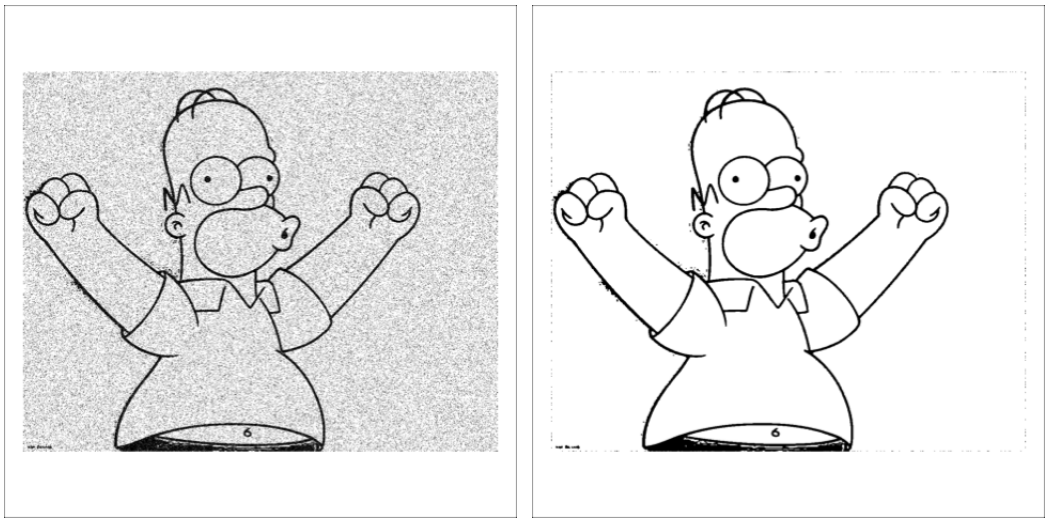
and if $y^+(m) = -1$,

$$\mu(x^+(m) = 1|x^-) = \frac{e^{2\beta M}(1-p)}{e^{2\beta M}(1-p) + e^{-2\beta M} p},$$

$$\mu(x^+(m)=-1|x^-)=\frac{e^{-2\beta M}p}{e^{2\beta M}(1-p)+e^{-2\beta M}p}.$$

It is then easy to sample from $\mu(.|x^-)$ (and also from $\mu(.|x^+)$. Let $(X_n)_{n\geq 0}$ be a sequence obtained by the Gibbs sampler algorithm (whose recurrence is described in Equation (8.1), the law of X_n is near μ (for $n\to+\infty$).

To manipulate images in R, we use thePNG[1] package. We start from the image on the left in Figure 8.2.3 (containing obviously some noise) and after a few iterations, we end up with the image on the right ($\beta=4$, $p=0.9$).



(A) Before (B) After

FIGURE 8.2.3.

8.3. Metropolis-Hastings algorithm

8.3.1. Description of the algorithm. We want to sample according to some law p^* (called the target law). I want to build a random sequence $(X_n)_{n\geq 0}$ which converges in law to p^*. Ingredient: a proposal kernel q, for all x, $x'\mapsto q(x'|x)$ is a probability density. We first describe the sequence algorithmically (how I would simulate it).

- Suppose I have X_n. I draw a proposal Y_{n+1} with law $q(\ldots|X_n)$.
- I flip a coin and I move to Y_{n+1} ($X_{n+1}=Y_{n+1}$) with probability $1\wedge\alpha(X_n,Y_{n+1})$ or I stay in X_n ($X_{n+1}=X_n$). Where:

$$\alpha(x,x')=\frac{p^*(x')q(x|x')}{p^*(x)q(x'|x)}.$$

When in X_n, the law of X_{n+1} depends only on X_n. So, (X_n) is "obviously" a Markov chain. Let us compute its kernel $k(x,x')=k(x'|x)$. We compute (first for $x'\neq x$)

$$\begin{aligned}\mathbb{P}(X_{n+1}=x'|X_n=x) &= \mathbb{P}(Y_{n+1}=x',\text{"move accepted"}|X_n=x)\\ &= \mathbb{P}(\text{"move accepted"}|Y_{n+1}=x',X_n=x)\mathbb{P}(Y_{n+1}=x'|X_n=x)\\ &= \min\big(1,\alpha(x,x')\big)\,q(x'|x)\\ &= \min\left(1,\frac{p^*(x')q(x|x')}{p^*(x)q(x'|x)}\right)q(x'|x).\end{aligned}$$

[1] see `https://iut-info.univ-reims.fr/users/nocent/R/#dimensions-de-limage` for a tutorial, in French

REMARK 8.2. If $p^*(x) = \lambda \times f(x)$ with λ unknown (we know p^* up to a normalizing constant). Then, there is classical method to sample from p^*. Observe that in our algorithm, we need only to compute α (where the λ's simplify).

REMARK 8.3. In the case where q is symmetric ($q(x'|x) = q(x|x')$ for all x, x', the algorithm is simply called "Metropolis".

THEOREM 8.4. *(Under some assumptions)*

- The chain (X_n) has invariant law p^*.
- $(X_n) \underset{n\to+\infty}{\longrightarrow} p^*$ in law
- $\frac{1}{n}\sum_{i=1}^n \varphi(X_i) \underset{n\to+\infty}{\longrightarrow} p^*(\varphi)$ almost surely ($p^*(\varphi) = \int \varphi(x) p^*(dx)$)

Proposal distribution. For a given target p^*, a proposal q is valid if it gives us a non-zero probability of moving to the states that have a non-zero probability in the target.

8.3.2. Gibbs sampling is a special case of MH (MH=Metropolis-Hastings). Imagine you are in $x = (x_1, x_2, \dots, x_D)$. We introduce a notation: $x_{-i} = (x_1, \dots, x_{i-1}, x_{i+1}, \dots, x_n)$ (physicist's notation: $(x_1, \dots, \widehat{x_i}, \dots, x_n)$). Instead of using a unique proposal, we use a sequence of proposals:

$$q_i(x'|x) = p(x'_i|x_{-i})\mathbb{1}_{\{x'_{-i}=x_{-i}\}}$$

(we modify only the component number i in x). Here p is the target law. We cycle through this sequence of proposals (there are D possible proposals). Each time, we make a proposition, we accept it with some probability:

$$\begin{aligned}
\alpha(x,x') &= \frac{p(x')q_i(x|x')}{p(x)q_i(x'|x)} \\
&= \frac{p(x'_i|x'_{-i})p(x'_{-i})p(x_i|x'_{-i})\mathbb{1}_{\{x'_{-i}=x_{-i}\}}}{p(x_i|x_{-i})p(x_{-i})p(x'_i|x_{-i})\mathbb{1}_{\{x'_{-i}=x_{-i}\}}} \\
&= \frac{p(x'_i|x_{-i})p(x_{-i})p(x_i|x_{-i})\mathbb{1}_{\{x'_{-i}=x_{-i}\}}}{p(x_i|x_{-i})p(x_{-i})p(x'_i|x_{-i})\mathbb{1}_{\{x'_{-i}=x_{-i}\}}} \\
&= 1
\end{aligned}$$

Which means we accept each proposal with probability 1. Let us give an example:

- I start with $x = (x_1, \dots, x_D)$. I sample $x'_1 \sim p(x_1|x_{-1})$. I get $(x'_1, x_2, x_3, \dots, x_n)$.
- ...
- Suppose I have $(x'_1, \dots, x'_k, x_{k+1}, \dots, x_D)$. Then I sample $x'_{k+1} \sim p(x_{k+1}|x'_1, \dots, x'_k, x_{k+2}, \dots, x_D)$.
- ...

8.3.3. Why MH works. Let us call $p(x'|x)$ the transition kernel of the Metropolis-Hastings sequence. We computed just above: $p(x'|x)$ for $x' \neq x$. For $x' = x$,

$$p(x|x) = 1 - \sum_{x' : x' \neq x} p(x'|x)\,.$$

THEOREM 8.5. *The target p^* is a stationary distribution for the Metropolis-Hastings chain (it is unique if the chain is irreducible)*

PROOF.

(1) Let us show that, for all x, x', $p^*(x)p(x'|x) = p^*(x')p(x|x')$ (we say that p^* is symmetric with respect to p). For $x' \neq x$:

$$\begin{aligned}
p^*(x)p(x'|x) &= p^*(x)q(x'|x)\min\left(1, \frac{p^*(x')q(x|x')}{p^*(x)q(x'|x)}\right) \\
&= \min(p^*(x)q(x'|x), p^*(x')q(x|x')) \\
&= \min\left(\frac{p^*(x)q(x'|x)}{p^*(x')q(x|x')} \times p^*(x')q(x|x'), p^*(x')q(x|x')\right)
\end{aligned}$$

$$\begin{aligned} &= p^*(x')q(x|x')\min\left(\frac{p^*(x)q(x'|x)}{p^*(x')q(x|x')},1\right) \\ &= p^*(x')p(x|x')\,. \end{aligned}$$

(2) Let us show now that p^* is actually stationary. For any φ:

$$\begin{aligned} \sum_{x,x'} p(x'|x)p^*(x)\varphi(x') &= \sum_{x,x'} p(x|x')p^*(x')\varphi(x') \\ &= \sum_{x'} p^*(x')\varphi(x') \\ &= p^*(\varphi)\,. \end{aligned}$$

□

For those who have seen a Markov chain course: when we have a stationary distribution for (X_n), we expect X_n to converge in law to this distribution.

REMARK 8.6. Another way to interpret the scheme. We propose moves with a kernel q. Suppose we are in x and propose a move in x'. The acceptance ratio is

$$\frac{p(x')q(x|x')}{p(x)q(x'|x)}\,.$$

If you forget about q, you can see that the ratio is big if $p(x') > p(x)$ (and small in the other case). If the ratio is big, we are more likely to accept the move. In a symmetric way, if we are in an area where $p(x)$ is big, we are not likely to accept the move. So, all in all, the chain will spend more time in places where $p(\dots)$ is big.

8.3.4. Deciphering example. Each text is written with a limited number of characters: {'a','b', ..., '!', '?', ...} (total of n characters). We replace each character by a number in $\{1,2,\dots,n\}$. A text can be transformed into a list of numbers (and transformed back into a text). This is exactly what is done by the ASCII standard code. A simple cipher method consists in changing these numbers (this might be the oldest cipher method in the world). Actually, we choose a permutation σ from $\{1,2,\dots,n\}$.

$$\underbrace{(a_1,a_2,\dots)}_{\substack{\text{original text represented}\\ \text{by a sequence of numbers}}} \longrightarrow \underbrace{(\sigma(a_1),\sigma(a_2),\dots)}_{\text{ciphered sequence}}$$

EXAMPLE 8.7. Imagine A is 1, B is 2, ... For the word "CAT", my list of numbers is: $(3,1,20)$. Imagine σ is a transposition (it exchanges two numbers): $\sigma(1) = 2$, $\sigma(2) = 1$ and $\sigma(k) = k$ for $k \geq 3$. My ciphered list for my word is: $(3,2,20)$.

I communicate the key of my cipher to my correspondent (namely: σ and σ^{-1}) and then my correspondent can read my messages.

The model is the following. A text is seen as a realization of a Markov chain whose state space is $\{1,\dots,n\}$. For example, I imagine that I have a chain (X_1,X_2,X_3) such that $X_1 = 3$, $X_2 = 1$, $X_3 = 20$. Suppose I have access to the transition matrix M. How can I get access to this? I simply take a really long text (usually a novel). This text is represented by a sequence of numbers : $(X_1(\omega),X_2(\omega),\dots,X_N(\omega))$. An easy approximation of $M(i,j)$ is

$$\begin{aligned} M(i,j) &= \mathbb{P}(X_{n+1} = j | X_n = i) \\ &= \frac{\mathbb{P}(X_{n+1} = j, X_n = i)}{\mathbb{P}(X_n = i)} \\ &\approx \frac{\left(\sum_{k=1}^{N-1} \mathbb{1}_{X_{k+1}=j}\mathbb{1}_{X_k=i}\right)/(N-1)}{\left(\sum_{k=1}^{N} \mathbb{1}_{X_{k=i}}\right)/N}\,. \end{aligned}$$

The underlying assumption is that the chain is stationary (meaning law of X_n does not depend on n, and consequently, the law of (X_n,X_{n+1}) does not depend on n).

I have a candidate s in $\mathfrak{S}_n$ (set of permutations of $\{1,2,\dots,n\}$) (I expect this guy to be σ^{-1}). I take my ciphered list $(b_1,b_2,\dots,b_N)$ and I look at the deciphering provided by s:

$$(s(b_1),\dots,s(b_N)).$$

I give s a score saying how much a good deciphering it is:

$$\Psi(s)=\prod_{k=1}^{N-1}M(s(b_k),s(b_{k+1})).$$

EXAMPLE 8.8. Suppose the ciphered text is $(25,26,1)$ ("YZA"). Suppose s is a transposition such that $s(25)=26$, $s(26)=25$ ($s(k)=k$ if $k\notin\{25,26\}$). Then my deciphered text is "ZYA". As the letter Z is rarely followed by Y (and Y is sometimes followed by A), the score of s is small.

My goal is to sample according to the law (target law)

$$\pi(s)=\frac{\Psi(s)}{\sum_{\sigma\in\mathfrak{S}_n}\Psi(s)}.$$

I will mostly get s with high score, meaning something almost readable. We run a MCMC (in the space $\mathfrak{S}_n$) with target law π and the proposal kernel I want (ideally, a proposal which allows to explore the whole space). Let us call Q the proposal. All I need is to be able to sample from $Q(f,\dots)$ for any f in $\mathfrak{S}_n$. Suppose, we have f in $\mathfrak{S}_n$. We follow the procedure.

- We sample $\{i,j\}$ uniformly in $\mathscr{P}_2(\{1,2,\dots,n\})$ ($\mathscr{P}_2(\dots)$ is the set of subsets of $\{1,2,\dots,n\}$ with cardinality 2).
- I define $f_{(i,j)}$ by

$$f_{(i,j)}(k)=\begin{cases}f(k) & \text{if } k\notin\{i,j\}\\ f(j) & \text{if } k=i\\ f(i) & \text{if } k=j.\end{cases}$$

- I jump in $f_{(i,j)}$.

We could check that Q is irreducible (meaning, that we can visit the whole set $\mathfrak{S}_n$) and $Q(f,g)=Q(g,f)$ for all g, f.

Remember the MCMC algorithm.

- Start with S_0 in $\mathfrak{S}_n$ (anywhere).
- When you are in S_t (at time t).
  - Propose R_{t+1} with law $Q(S_t,\dots)$.
  - Compute the ratio

$$\begin{aligned}\alpha(S_t,R_{t+1}) &= \frac{\pi(R_{t+1})Q(R_{t+1},S_t)}{\pi(S_t)Q(S_t,R_{t+1})}\\ &= \frac{\Psi(R_{t+1})Q(R_{t+1},S_t)}{\Psi(S_t)Q(S_t,R_{t+1})}\\ &= \frac{\Psi(R_{t+1})}{\Psi(S_t)}.\end{aligned}$$

  - Draw $U\sim\mathscr{U}([0,1])$. Move to $S_{t+1}=R_{t+1}$ if

$$U\le\min(1,\alpha(S_t,R_{t+1}))$$

    (stay in S_t otherwise).

What we know from the course is that, if we run this MCMC "long enough", we get S_n (approximatively) of law π.

Starting from a ciphered text, we get the following after 5000 iterations

> all me shmael. ome years agonever mind how long preciselyhaving little or no money in my purse, and nothing particular to interest me on shore, thought would sail about a little and see the watery part of the world. t is a way have of driving off the spleen and regulating the circulation. henever find myself growing grim about the mouth; whenever it is a damp, dri"ly ovember in my soul; whenever find myself

> involuntarily pausing before coffin warehouses, and bringing up the rear of every funeral meet; and especially whenever my hypos get such an upper hand of me, that it requires a strong moral principle to prevent me from deliberately stepping into the street, and methodically knocking peoples hats offthen, account it high time to get to sea as soon as can. his is my substitute for pistol and ball. ith a philosophical flourish ato throws himself upon his sword; quietly take to the ship. here is nothing surprising in this. f they but knew it, almost all men in their degree, some time or other, cherish very nearly the same feelings towards the ocean with me

So the original text might very well be

> Call me Ishmael. Some years ago—never mind how long precisely—having little or no money in my purse, and nothing particular to interest me on shore, I thought I would sail about a little and see the watery part of the world. It is a way I have of driving off the spleen and regulating the circulation. Whenever I find myself growing grim about the mouth; whenever it is a damp, drizzly November in my soul; whenever I find myself involuntarily pausing before coffin warehouses, and bringing up the rear of every funeral I meet; and especially whenever my hypos get such an upper hand of me, that it requires a strong moral principle to prevent me from deliberately stepping into the street, and methodically knocking people's hats off—then, I account it high time to get to sea as soon as I can. This is my substitute for pistol and ball. With a philosophical flourish Cato throws himself upon his sword; I quietly take to the ship. There is nothing surprising in this. If they but knew it, almost all men in their degree, some time or other, cherish very nearly the same feelings towards the ocean with me.
>
> — Herman Melville, *Moby Dick*, 1851. —

8.4. Advantages/disadvantages of MCMC methods

Advantages:

- applicable even when we can't directly draw samples,
- works for complicated distributions in high-dimensional spaces, even when we don't know where the regions of high probability are,
- relatively easy to implement,
- fairly reliable.

Disadvantages:

- can be very difficult to assess accuracy and evaluate convergence, even empirically,
- can be slow (need a lot of samples to be precise).

8.5. Examples in R

8.5.1. Volleyball. We are going to analyze the 2016 Big 10 conference volleyball results[2].

```
big10<-read.csv("datasets/volley.csv")
n=nrow(big10)
head(big10)
```

```
##       home            road home.win home.sets road.sets     date
## 1 Illinois          Rutgers        1         3         0 09/24/16
## 2 Illinois        Nebraska         0         0         3 09/28/16
## 3 Illinois Michigan State          1         3         0 10/12/16
## 4 Illinois   Northwestern          1         3         0 10/15/16
## 5 Illinois         Indiana         1         3         0 10/21/16
## 6 Illinois          Purdue         0         2         3 10/22/16
```

[2] https://www.stat.umn.edu/geyer/3701/notes/mcmc-bayes.html

Each row of this data frame is data for one match. We are only going to use the first three columns in our analysis. These are home team, away team, and whether the home team won.

The response variable (what is random) is zero-or-one-valued (1 for home team win, 0 for home team loss = away team win). Hence logistic regression is appropriate. We will use what is called a Bradley-Terry model with home court advantage. For a match in which team i is the home team and team j is the away team, the response is a Bernoulli variable with parameter $\text{sigm}(\theta_{i,j})$, where

$$\theta_{i,j} = \beta_i - \beta_j + \gamma,$$

β_i represents the "strength" of team i, β_j represents the "strength" of team j and γ measures home court advantage. As the $\theta_{i,j}$'s are invariant if we translate all the β_i's (only their relative positions are important), we need to add more information in our model to fix the β_i's. So we anchor β_1 to zero ($\beta_1 = 0$).

We want to estimate the parameters (β_i), γ. In particular, we would like to have an idea of what are the three best teams.

We extract the team names.

```
teams <- unique(big10$home)
p=length(teams)
```

Each team has now a number (its rank in the `teams` list there are $p = 14$ teams). For each match, we want a line in which the i-th term is one of team number i is the home team, and minus one if team number i is the road team. We stack these lines together to create a matrix.

```
modmat <- matrix(0,nrow=n,ncol=p)
i=0
for (team in teams)
{
        i <- i+1
        foo <- rep(0, nrow(big10))
        foo[big10$home == team] <- 1
        foo[big10$road == team] <- (-1)
        foo <- matrix(foo,n,1)
        modmat[,i]<-foo
}
```

We will put our parameters $\beta_2, \beta_3, \ldots$ and γ in a single vector named `beta`. We need a function that evaluates the log-likelihood (on any `beta`). If X is Bernoulli of parameter p, then the density of X is

$$x \in \mathbb{R} \mapsto p\delta_1(x) + (1-p)\delta_0(x).$$

As X is discrete-valued and takes the values 0 and 1, the density is a density with respect to the measure $\delta_0+\delta_1$. Suppose now that we observe X, a Bernoulli variable, the log-likelihood of parameter p, knowing X is thus the following:

$$p \in [0,1] \mapsto \log(p)\mathbb{1}_{X=1} + \log(1-p)\mathbb{1}_{X=0} = X\log(p) + (1-X)\log(1-p).$$

Suppose now, we have independent Bernoulli variables $X_1, \ldots, X_n$ with parameters $p_1, \ldots, p_n$, the log-likelihood is

$$(p_1,\ldots,p_n) \mapsto \sum_{i=1}^{n} X_i \log(p_i) + (1-X_i)\log(1-p_i).$$

So, we build our log-likelihood as follows (this is the likelihood of the parameters, knowing the outcomes of the matches).

```
logL <- function(beta)
{
        # add beta_1
        beta=c(0,beta)
        # compute beta_i - beta_j for each match
```

```
        eta <- drop(modmat %*% beta[1:p])+matrix(beta[p+1],n,1)
        # plogis is the sigmoid function
        probas <- plogis(eta,0,1)
        big10$probas <- probas
        #log1p(x)=1-log(x)
        big10$log<-ifelse(big10$home.win==1,log(big10$probas),
                                                log1p(-big10$probas))
        return(sum(big10$log))
}
```

We want to use informative priors, but unlike most Bayesian analyses, we do not want the prior to favor any team over any other team. We want to let the data speak for itself. We imagine that each team played at a neutral site (no home court advantage) against an imaginary team whose ability parameter β is fixed at zero. Thus all teams are now measured relative to this imaginary team. And we imagine that each real team had one win and one loss against this imaginary team. This gives a prior

$$\prod_{i=2}^{p} \text{sigm}(\beta_i) \times (1 - \text{sigm}(\beta_i)).$$

But we also need a prior for γ (home court advantage). We will suppose that two teams of equal ability (equal β's) played two games, and the home team won one and lost one. That gives us another factor

$$\text{sigm}(\gamma)(1 - \text{sigm}(\gamma)).$$

Thus the log-prior (for all parameters) is

$$\log(\text{sigm}(\gamma)) + \log(1 - \text{sigm}(\gamma)) + \sum_{i=1}^{p} \log(\text{sigm}(\beta_i)) + \log(1 - \text{sigm}(\beta_i)).$$

```
logPrior <- function(beta)
{
        beta <- c(0,beta)
        #log1p(x)=log(1+x) (more precise computation)
        z <- sum( log(plogis(beta,0,1))+log1p(-plogis(beta,0,1)))
        return(z)
}
```

Our log-posterior is then

```
logPosterior <- function(beta)
{
        return(logL(beta)+logPrior(beta))
}
```

We want to sample from this density, so this is our target law.

We use the `metrop` command.

```
set.seed(42) # for reproducibility
library(mcmc)
out <- metrop(logPosterior,rep(0,p),nbatch=100,blen=100,scale=0.1)
```

Here, we have specified

- the initial value for the parameter vector (`rep(0,p)`, a vector of zeros),
- the logarithm of the target density (`logPosterior`),
- the number of batches (`nbatch`), the algorithm retains the values of the MCMC chain at the end of each batch only (do not retain all the values),
- the length of each batch (`blen`),
- the proposal is made of i.i.d. Gaussians, we specify their standard deviation in `scale=...`.

The result `out` is an object of class `mcmc`, subclass `metrop`, which is a list containing a lot of components. Conventional wisdom says that one should adjust the acceptance rate (the proportion of the time the algorithm accepts a proposed move) to be about 25%. We can play with the `scale` parameter in order to get an acceptance rate of 25%. The parameter `scale=0.35` seems to get us an acceptance rate close enough to 25%.

```
out <- metrop(logPosterior,rep(0,p),nbatch=100,blen=100,scale=0.35)
out$accept
```

```
[1] 0.2552
```

It is clear from the way the algorithm works that, as the target p^* is continuous, one can make the acceptance rate as close to one as one pleases by making the steps very, very small. But that is a terrible idea, because it will take a very long time to get from one part of the sample space to another.

In the opposite direction, one could try to take very big steps (make the scale large). But when x looks like a sample from the equilibrium distribution, so $p^*(x)$ is moderately large, $x+y$ will be way out in the tails of the distribution and the ratio $p^*(x+y)/p^*(x)$ will be very, very small. So almost no steps are accepted.

Thus the scale needs to be adjusted so it is not too large and not too small. It is a Goldilocks problem — so-called because of one bowl of porridge being too hot, the next too cold, and the third just right and one of the beds being too hard, the next too soft, and the third just right, and so forth. We don't want the scale to be too large or too small. We want it to be just right (but unfortunately we have no way to recognize "just right").

Diagnosis plots. MCMC has diagnostic plots. Like for linear regression, the plots do not find all problems but only gross problems that jump out at you from certain plots.

If we plot the time series of any component of the Markov or any function of it (acceptance indicators, for example) or their batch means, the time series should look stationary.

```
plot(ts(out$batch[ , 14]), main="Batch Means for Home Advantage Coefficient")
```

gives us Figure 8.5.1

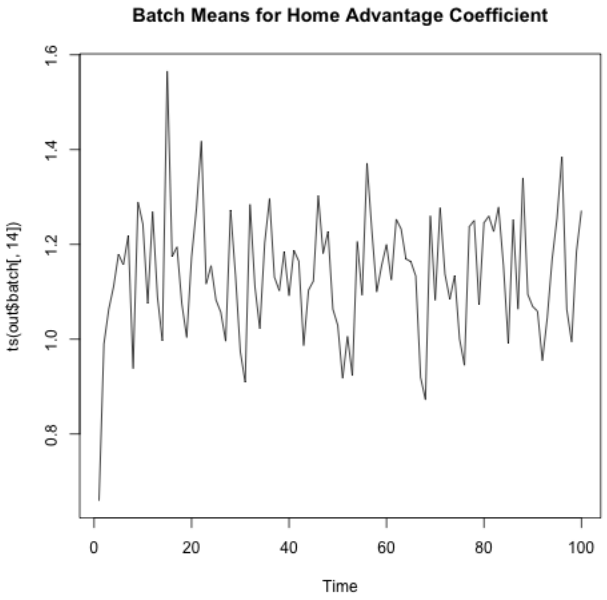


FIGURE 8.5.1. Batch Means for Home Advantage Coefficient

```
plot(ts(out$batch[ , 1]), main="Batch Means for Coefficient for Indiana")
```

gives us Figure 8.5.2 and so on.

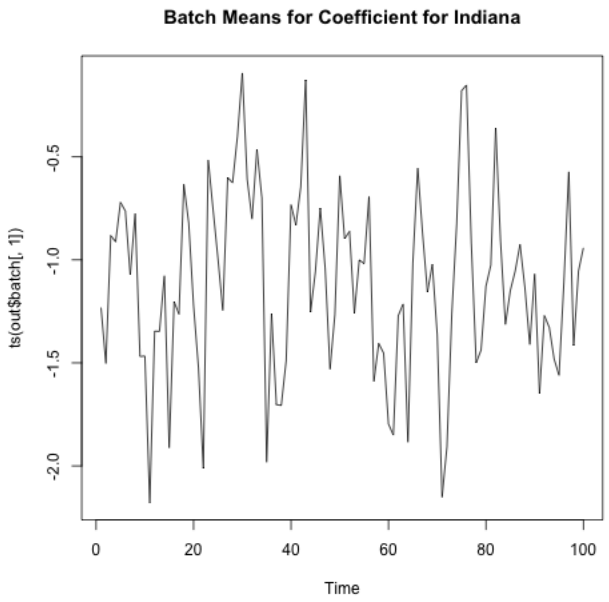


FIGURE 8.5.2. Batch Means for Coefficient for Indiana

The autocorrelation plots give the autocorrelation of the time series of batch means (the mean over a batch). Autocorrelation is correlation of the time series with itself at a later time. There should not be significantly nonzero correlation except for lag zero. Otherwise the batch length is not long enough.

```
acf(ts(out$batch[ , 14]), main="Autocorrelation of Home Advantage Coefficient")
```

gives us Figure 8.5.3,

```
acf(ts(out$batch[ , 1]), main="Autocorrelation of Coefficient for Indiana")
```

gives us Figure and so on. It seems that at least some of the lag one autocorrelations are significant (the blue dashed lines are supposed to be cutoff points for 0.05 level tests of the null hypothesis that the true unknown autocorrelation is zero. Hence we should double the batch length (or more) to be safe. For each β_i, we can plot an histogram of the a posteriori density (see Figure 8.5.5).

The issue we will address now is which team is best, and more specifically what is the probability that each team is best.

We have applied Bayes' rule already. We have sampled the posterior. But we have not examined the function of those parameters that addresses this question. In order to calculate the probabilities of these events, we need to output the indicators of these events.

```
outFunction <- function(beta)
{
        beta <- beta[-p]
        beta=c(0,beta)
        return(as.numeric(beta == max(beta)))
}
```

We use

```
mout <- metrop(logPosterior,rep(0,p),nbatch=100,scale=0.1,outfun=outFunction)
```

Specifying the `outfun` parameters makes the program to compute the batch means of `outfun` (it writes them in `out$batch`). We then put names on the columns of `out$batch` and take the means.

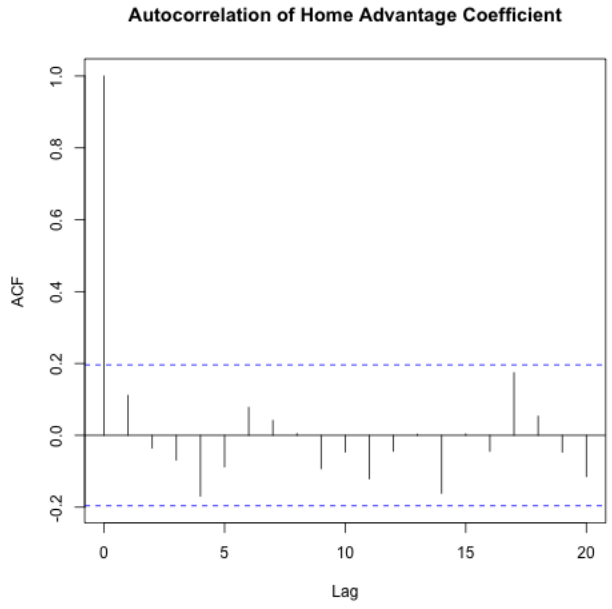


FIGURE 8.5.3. Autocorrelation for home advantage coefficient

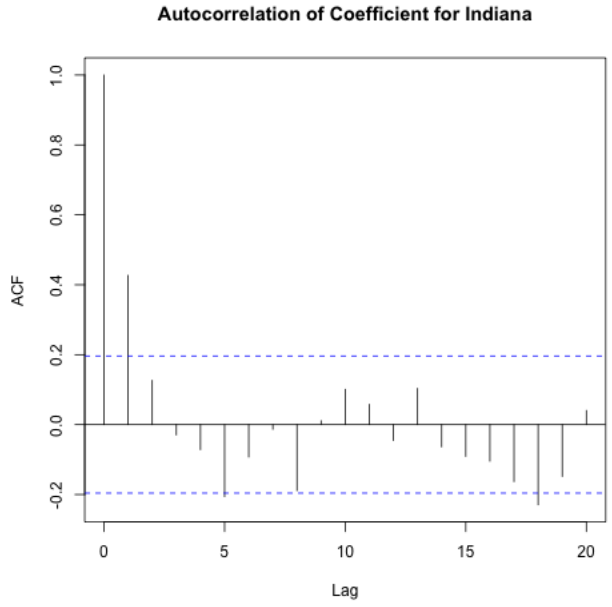
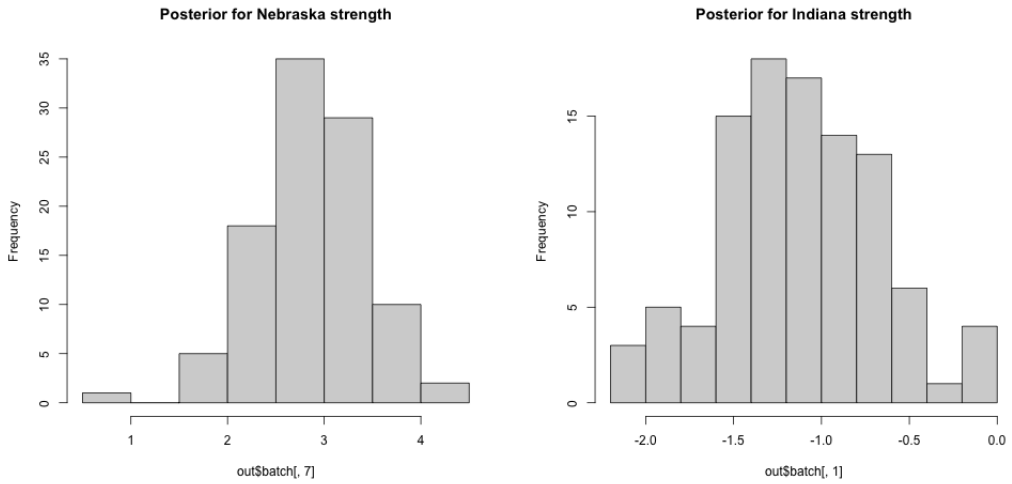


FIGURE 8.5.4. Autocorrelation of Coefficient for Indiana

```
teamBatch<-data.frame(mout$batch)
colnames(teamBatch) <- teams
colMeans(teamBatch)
```

```
  Illinois          Indiana             Iowa         Maryland
0.00005          0.00005          0.00005          0.00005
Michigan  Michigan State        Minnesota         Nebraska
```



(A) Nebraska (B) Indiana

FIGURE 8.5.5. Posteriors for "strength of a team".

```
0.00045         0.00430         0.29165         0.50210
Northwestern      Ohio State      Penn State         Purdue
0.00005         0.00005         0.01555         0.00020
Rutgers       Wisconsin
0.00005         0.18605
```

Only the estimated top 3 teams have appreciable probability of having been the actual best.

8.5.2. Pareto model. Distributions of sizes and frequencies often tend to follow a "power law" distribution. Here are a few examples of data which have been claimed to follow this type of distribution:

- wealth of individuals
- size of oil reserves
- size of cities
- word frequency
- returns on stocks
- size of meteorites.

The Pareto distribution with shape $\alpha > 0$ and scale $c > 0$ has the density

$$\text{Pareto}(x|\alpha, c) = \frac{\alpha c^\alpha}{x^{\alpha+1}} \mathbb{1}_{x>c}.$$

This is referred to as a power law distribution, because the density is proportional to x raised to a power. Notice that c is a lower bound on the observed values. In this example, we'll see how Gibbs sampling can be used to perform inference for α and c. We download the data[3][4] (sizes of North-Carolina cities), and keep only the first fifty lines.

```
data<-read.csv("datasets/north-carolina-cities.csv")
data=data[1:50,]
head(data)
```

```
rank            name   pop2021  pop2010 change    density
1          Charlotte   912096   738444 0.2352  1146.2179
```

[3] http://jwmi.github.io/BMS/chapter6-gibbs-sampling.pdf

[4] http://www2.stat.duke.edu/~rcs46/modern_bayes17/lecturesModernBayes17/lecture-7/07-gibbs.pdf

2	Raleigh	483579	406353	0.1900	1279.8377
3	Greensboro	301094	269598	0.1168	900.7132
4	Durham	287865	230727	0.2476	990.4180
5	Winston–Salem	250765	230022	0.0902	730.0950
6	Fayetteville	213475	208337	0.0247	557.4881

The parameter α tells us the scaling relationship between the size of cities and their probability of occurring. Let $\alpha = 1$. Density looks like $1/x^{\alpha+1} = 1/x^2$. So cities with 10,000–20,000 inhabitants occur roughly $10^{\alpha+1} = 100$ times as frequently as cities with 100,000–110,000 inhabitants.The parameter c represents the cutoff point—any cities smaller than this were not included in the dataset.

For simplicity, let's use an (improper) default prior:

$$p(\alpha, c) \propto \mathbb{1}_{\alpha,c>0}.$$

Recall:

- An improper/default prior is a non-negative function of the parameters which integrates to infinity.
- Often (but not always!) the resulting "posterior" will be proper.
- It is important that the "posterior" be proper, since otherwise the whole Bayesian framework breaks down.

Let us call $x_{1:n}$ our list of city sizes. We assume this is an i.i.d. sample from a Pareto law (we want to estimate the parameters of this law). We compute the posterior:

$$\begin{aligned} p(\alpha, c|x_{1:n}) &\propto \mathbb{P}(x_{1:n}|\alpha, c)\, p(\alpha, c) \\ &= \prod_{i=1}^{n} \frac{\alpha c^\alpha}{x_i^{\alpha+1}} \mathbb{1}_{x_i>c} \times \mathbb{1}_{\alpha,c>0} \\ &= \frac{\alpha^n c^{n\alpha}}{\left(\prod_{i=1}^n x_i\right)^{\alpha+1}} \mathbb{1}_{c<x_*} \mathbb{1}_{\alpha,c>0}, \end{aligned}$$

where $x_* = \min_{1\le i\le n} x_i$. We need now to compute $p(\alpha|c, x_{1:n})$. We have:

$$\begin{aligned} p(\alpha|c, x_{1:n}) &= \frac{p(\alpha, c|x_{1:n})}{p(c|x_{1:n})} \\ (c \text{ is fixed}) \quad &\propto \frac{\alpha^n c^{n\alpha}}{\left(\prod_{i=1}^n x_i\right)^{\alpha+1}} \mathbb{1}_{\alpha>0} \\ &= \frac{\alpha^n \exp(n\alpha \log(c))}{\exp((\alpha+1)\sum_{i=1}^n \log(x_i))} \mathbb{1}_{\alpha>0} \\ &\propto \Gamma\left(\alpha|n+1, \sum_{i=1}^n \log(x_i) - n\log(c)\right). \end{aligned}$$

This is the density (in α) of the Gamma distribution with shape parameter $n+1$ and rate parameter $\sum_{i=1}^n \log(x_i) - n\log(c)$. The law $p(\alpha|c, x_{1:n})$ is proportional to a Gamma law but are they are both laws, they are equal.

We also compute:

$$\begin{aligned} p(c|\alpha, x_{1:n}) &= \frac{p(\alpha, c|x_{1:n})}{p(\alpha|x_{1:n})} \\ (\alpha \text{ is constant}) \quad &\propto p(\alpha, c|x_{1:n}) \\ &\propto c^{n\alpha} \mathbb{1}_{0<c<x_*}. \end{aligned}$$

We introduce the probability density function (with parameters a, b):

$$\text{Mono}(x|a, b) = \frac{a x^{a-1}}{b^a} \mathbb{1}_{0<x<b}.$$

The law $p(c|\alpha, x_{1:n})$ follows this Mono law (with parameters $a = n\alpha+1$, $b = x_*$). We need to sample from this law. We will do it using the inverse of the cumulative distribution (called $F(\ldots|a, b)$). We have (for x

in $[0,b]$)

$$\begin{aligned} F(x|a,b) &= \int_0^x \frac{at^{a-1}}{b^a}\mathbb{1}_{0<t<b}dt \\ &= \left[\frac{t^a}{b^a}\right]_0^x = \frac{x^a}{b^a}. \end{aligned}$$

We can compute the inverse of F:

$$F^{-1}(y) = by^{1/a}.$$

So, if $U \sim \mathcal{U}([0,1])$, then $bU^{1/a}$ has the law Mono(a,b). The following function samples from Mono(a,b)

```
mono <- function(a,b)
{
        u=runif(1,0,1)
        return(b*u^(1/a))
}
```

Our algorithm takes the following form in R:

```
n=nrow(data)
N=1000
alpha=c(1)
c=c(500)
for (i in 1:N)
{
        foo <- rgamma(1,shape=n+1,rate=Sum-n*log(c[length(c)]))
        alpha= c(alpha,foo)
        foo <- mono(n*alpha[length(alpha)]+1,Min)
        c=c(c,foo)
}
```

We can plot the sequences `alpha` and `c` (after some burn-in time). See the results in Figure 8.5.6.

```
par(mfrow=c(2,1))
plot(seq(100,N,1),alpha[100:N])
plot(seq(100,N,1),c[100:N])
```

To check if we have reached stationarity, we can plot the running averages (see Figure , this time for $N = 1000$).

```
library(igraph)
par(mfrow=c(2,1))
plot.ts(running_mean(alpha,binwidth=50),ylab='moving average for alpha')
plot.ts(running_mean(c,binwidth=50),ylab='moving average for c')
```

This seems OK, no need to take a bigger N.

We can plot histograms approximating the posterior densities of α, c (see Figure 8.5.8), this time for $N = 10000$.

```
hist(alpha,freq=FALSE)
abline(v=quantile(alpha[100:N],c(0.9)),col="blue")
abline(v=quantile(alpha[100:N],c(0.1)),col="blue")
hist(c,freq=FALSE)
abline(v=quantile(c[100:N],c(0.9)),col="blue")
abline(v=quantile(c[100:N],c(0.1)),col="blue")
```

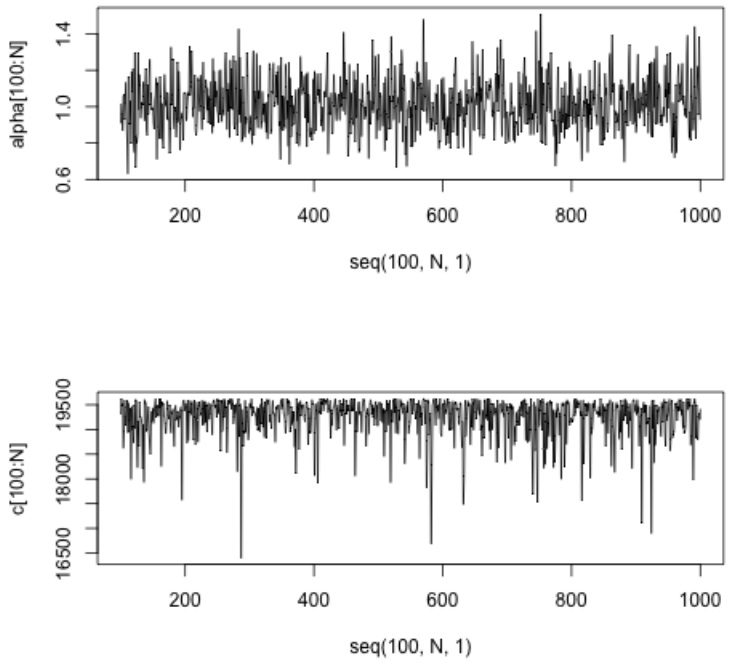


FIGURE 8.5.6. Traceplots

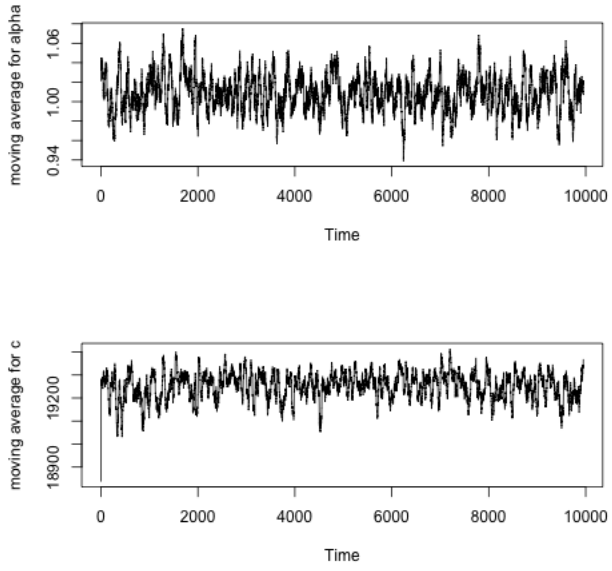
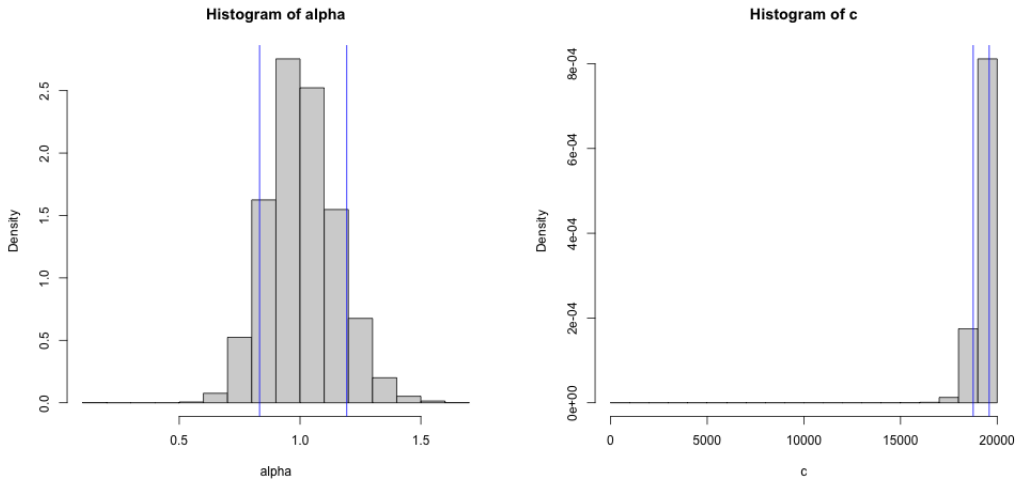


FIGURE 8.5.7. Moving averages

We have added blue lines materializing the 90% confidence intervals.

To check if our estimation is good, we will plot the survival functions (survival function=1-cumulative distribution function). We fix $\alpha = 1$, $c = 19243$. The survival function for the Pareto law with parameters



(A) α (B) c

FIGURE 8.5.8

α, c is simply (for $x \geq c$)

$$S(x) = \left(\frac{c}{x}\right)^{\alpha}.$$

We will plot it on a log-log plot (it will appear as a line with slope $-\alpha$). On the same graph, we plot the empirical survival function of the posterior. We have a sample $x_{1:n}$ of city sizes . This empirical survival function is simply:

$$x \mapsto \frac{1}{n}\sum_{i=1}^{n} \mathbb{1}_{x_i > x}.$$

See the result in Figure 8.5.9

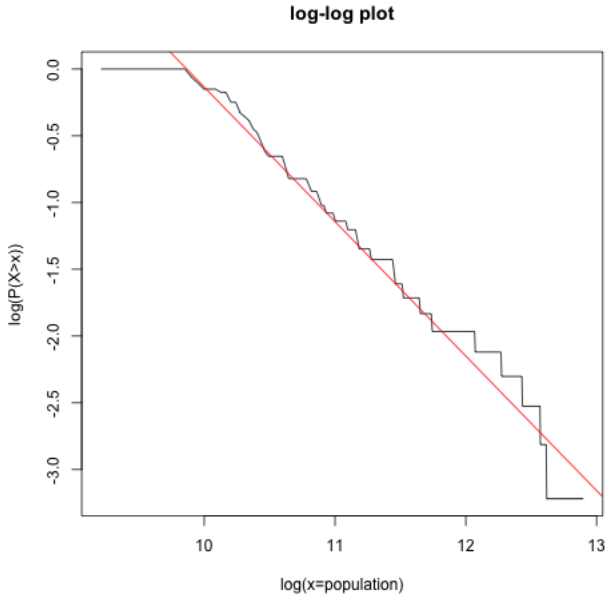


FIGURE 8.5.9. Log-log plot of the survival function.

8.5.3. World Cup problem ([AH20]) (activity). In the 2018 FIFA World Cup final, France defeated Croatia 4 goals to 2. Based on this outcome:

- How confident should we be that France is the better team?
- If the same teams played again, what is the chance France would win again?

To answer these questions, we have to make some modeling decisions.

- First, we assume that for any team against another team there is some unknown goal-scoring rate, measured in goals per game, which we denote λ.
- Second, we assume that a goal is equally likely during any minute of a game. So, in a 90 minute game, the probability of scoring during any minute is $\lambda/90$.
- Third, we assume that a team never scores twice during the same minute.

We expect the number of goals scored in a game to follow a Poisson distribution (with parameter λ, denoted by $\mathscr{P}(\lambda)$). Remember, if the number of goals scored in a game follows a Poisson distribution with a goal-scoring rate λ, the probability of scoring k goals is

$$\frac{\lambda^k e^{-\lambda}}{k!}$$

for any non-negative value of k.

We need now to choose a prior. If you have ever seen a soccer game, you have some information about λ. In most games, teams score a few goals each. In rare cases, a team might score more than 5 goals, but they almost never score more than 10. We will use a Gamma distribution. Remember a variable X follows a Gamma law with parameters k (shape), θ (scale) ($X \sim \Gamma(k,\theta)$) if it has the following density

$$x \mapsto f(x;k,\theta) = \frac{x^{k-1}e^{-\frac{x}{\theta}}}{\Gamma(k)\theta^k}\mathbb{1}_{x>0},$$

where Γ is the Euler Gamma function (we do not need its definition here). The variable X has then the mean $k\theta$. We choose $k = 3/2$ and $\theta = 1$.

We have two variables X_1, X_2, respectively of laws $\mathscr{P}(\lambda_1)$, $\mathscr{P}(\lambda_2)$. They represent the goals scored by France and Croatia (the outcome of the match is $X_1 = 4$, $X_2 = 2$). We want to compute the posterior

$$p(\lambda_1,\lambda_2|X_1,X_2) \propto p(X_1,X_2|\lambda_1,\lambda_2)p(\lambda_1,\lambda_2).$$

For the prior $p(\lambda_1,\lambda_2)$, we have chosen

$$p(\lambda_1,\lambda_2) = f(\lambda_1;k,\theta)f(\lambda_2;k,\theta).$$

Now, the likelihood $p(X_1,X_2|\lambda_1,\lambda_2)$ is

$$p(X_1,X_2|\lambda_1,\lambda_2) = \frac{\lambda_1^4 e^{-\lambda_1}}{4!} \times \frac{\lambda_2^2 e^{-\lambda_2}}{2!}.$$

(1) Build a Metropolis algorithm that samples from the law $p(\lambda_1,\lambda_2|X_1,X_2)$. Hints: We need to know the densities only up to a normalizing constant (i.e. we do not need to know the function Γ). The `metrop` command of `R` will make proposals with a Gaussian kernel, in the whole real line. So, your Metropolis chain should represent the logarithms (μ_1,μ_2) of (λ_1,λ_2) (the working posterior is thus $(\mu_1,\mu_2) \mapsto \lambda_1\lambda_2 p(\lambda_1 = e^{\mu_1}, \lambda_2 = e^{\mu_2}|X_1,X_2)$).

(2) Run the diagnosis tests we have seen in Section 8.5.1 (the volleyball example): acceptance rate, trajectory of batch means, autocorrelation. You might adjust the `scale` parameter in `metrop` at this point.

(3) What is the probability that France is the better team?

(4) If the same teams played again, what is the chance France would win again? Hints: We suppose that if the score is a tie, France has a 50% probability to win the match. When we have two independent variables $X \sim \mathscr{P}(\lambda)$, $Y \sim \mathscr{P}(\mu)$, we can compute $\mathbb{P}(X > Y)$ with a simple Monte-Carlo (say, with $N = 1000$ and independent samples $(X_1,Y_1),\dots,(X_N,Y_N)$ having the same law as (X,Y))

$$\mathbb{P}(X > Y) \approx \frac{1}{N}\sum_{i=1}^{N} \mathbb{1}_{X_i > Y_i}.$$

8.5.4. Summary of useful R commands. We use the following command to run a Metropolis chain.

```
out <- metrop(logPosterior,rep(0,p),nbatch=100,blen=100,scale=0.1)
```

Here, we have specified

- the initial value for the parameter vector (`rep(0,p)`, a vector of zeros),
- the logarithm of the target density (`logPosterior`),
- the number of batches (`nbatch`), the algorithm retains the values of the MCMC chain at the end of each batch only (do not retain all the values),
- the length of each batch (`blen`),
- the proposal is made of i.i.d. Gaussians, we specify their standard deviation in `scale=...`.

CHAPTER 9

Clustering

Clustering is nothing but grouping. We are given some data, we have to find some patterns in the data and group similar data together to form clusters . This is the basis of clustering. This is done with the help of euclidean distance.

For example:

- An athletic club might want to cluster their runners into 3 different clusters based on their speed (1 dimension).
- A company might want to cluster their customers into 3 different clusters based on 2 factors : Number of items brought, no of items returned (2 dimensions).

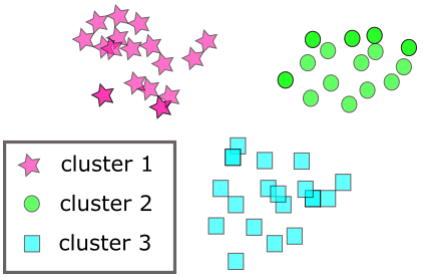


FIGURE 9.0.1. Clustering

9.1. K-means clustering

We have N points: $x_1, \ldots, x_N$ in $\mathbb{R}^D$. We want to group them in clusters (according to their distances within one another). We usually use the euclidean distance but one could do otherwise.

Suppose we have K clusters. For each n in $\{1,2,\ldots,N\}$, we have labels $z_1^{(n)}, \ldots, z_K^{(n)}$ such that

$$z_i^{(n)} = \begin{cases} 1 & \text{if } x_n \text{ in cluster } i\,, \\ 0 & \text{if } x_n \text{ not in cluster } i\,. \end{cases}$$

It is worth noting that, for all n, $\sum_{k=1}^{K} z_k^{(n)} = 1$. We define the mean of the k-th cluster (this could also be seen as the the center of mass) by

$$\mu_k = \frac{\sum_{n=1}^{N} z_k^{(n)} x_n}{\sum_{n=1}^{N} z_k^{(n)}}\,. \tag{9.1}$$

We suppose each point x_n is assigned to the the cluster k such that μ_k is the closest to x_n, in other words

$$\forall n\,,\, z_k^{(n)} = 1 \Leftrightarrow (x_n - \mu_k)^T (x_n - \mu_k) \geq (x_n - \mu_i)^T (x_n - \mu_i)\,(\forall i)\,.$$

We want to end up in such a configuration. To this purpose, we use an iterative algorithm.

You have to choose K by yourself. Start with initial random values $\mu_1, \ldots, \mu_K$ (see Figure , an example with $K = 2$).

(1) Assignment step/expectation step. For each x_n, find the k minimizing $(x_n - \mu_k)^T (x_n - \mu_k)$ and set $z_k^{(n)} = 1$ ($z_j^{(n)} = 0$ for $j \neq k$).

(2) If all the assignments ($z_k^{(n)}$) are unchanged (with respect to the previous state) → STOP,

(3) Update step/maximization step. Update μ_k with Equation (9.1) (meaning you compute a new μ_k using the above equation).
(4) Return to step 1.

Let us see on an example the algorithm in action. We start with Figure 9.1.1. We have two centroids placed randomly on the left (the stars). For all the yellow points, we find the closest centroid. Then we assign each point to a cluster (we color it with the same color as the closest centroid). In Figure 9.1.2,

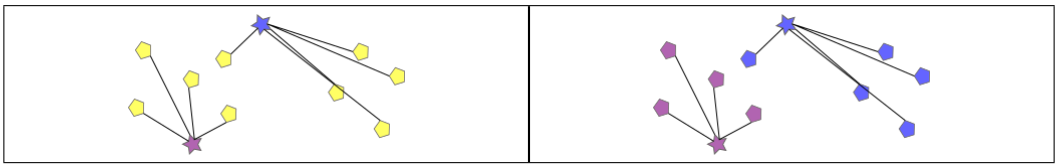


FIGURE 9.1.1. K-means algorithm (a)

we erase the previous centroids, add new centroids defined as centers of mass of the clusters (left of the Figure). Then, always on the left, we find the closest centroid for each point. And, as we see on the right of Figure 9.1.2, we color each point accordingly to the closest centroid. In Figure 9.1.3, we have erased

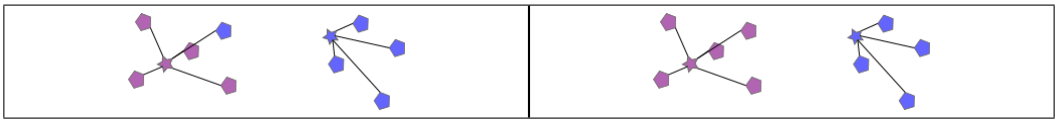


FIGURE 9.1.2. K-means algorithm (b)

the previous centroids and computed new centroids as barycentres of the clusters. For each point, the closest centroid is one with the same color as the point. So the algorithm stops there.

FIGURE 9.1.3. K-means algorithm (c)

Why does it work?

We are interested in a cost function:

$$F((\mu_k)_{1\leq k\leq K},(z_k^{(n)})_{1\leq k\leq K,1\leq n\leq N}) = \sum_{n=1}^{N}\sum_{k=1}^{K} z_k^{(n)}\|x_n-\mu_k\|^2 .$$

Step 1 of the above algorithm assigns a cluster number to each x_n. Let us call $y_k^{(n)}$ for the labels after the assignment step. We have

$$F((\mu_k),(z_k^{(n)})) \quad \geq \quad F((\mu_k),(y_k^{(n)})) .$$

Step 3 changes the (μ_k). Let us write (μ_k) for the means (or centers of mass) before the update step and (ν_k) for the means after the update step. The labels $(z_k^{(n)})$ are fixed. We look at the following function (from $(\mathbb{R}^d)^K$ to $\mathbb{R}$) (also called SSE)

$$\begin{aligned}\Phi(m_1,\dots,m_K) &= F((m_k),(z_k^{(n)}))\\ &= \sum_{n=1}^{N}\sum_{k=1}^{K} z_k^{(n)}(x_n-m_k)^T(x_n-m_k)\\ &= \sum_{n=1}^{N}\sum_{k=1}^{K} z_k^{(n)}\sum_{d=1}^{D}(x_{n,d}-m_{k,d})^2 .\end{aligned}$$

Let us look for a critical point of $\nabla\Phi$. We compute for all k, d:

$$\frac{\partial\Phi}{\partial m_{k,d}}(m_1,\dots,m_K) = \sum_{n=1}^{N}\sum_{k=1}^{K} z_k^{(n)} \times 2(m_{k,d} - x_{n,d})\,.$$

And so, if there is a critical point (where all these partial derivatives vanish), it is such that (for all d, k).

$$\sum_{n=1}^{N} z_k^{(n)} m_{k,d} = \sum_{n=1}^{N} z_k^{(n)} x_{n,d}\,,$$

So, there is only one critical point and it satisfies

$$\sum_{n=1}^{N} z_k^{(n)} m_k = \sum_{n=1}^{N} z_k^{(n)} x_n\,,$$

$$m_k = \frac{\sum_{n=1}^{N} z_k^{(n)} x_n}{\sum_{n=1}^{N} z_k^{(n)}}\,.$$

We also observe that Φ is convex (it is a sum of squares ...). So this critical point is an absolute minimum. And now, let us call (ν_k) the critical point of Φ. If we look at Equation (9.1) above, we see that what we choose for the update we do in Step 3 is the the minimum of Φ. So

$$F((\mu_k),(z_k^{(n)})) = \Phi((\mu_k)) \geq \Phi((\nu_k)) = F((\nu_k),(z_k^{(n)}))\,.$$

At each step of the algorithm, F can only decrease. As the function F is bounded by below (it is nonnegative), the algorithm stops (if it stops, because we do not prove that it stops) in a local minimum of F (which we hope is the absolute minimum of F).

9.1.1. Choosing the number of clusters K. The cost function F decreases when K increases. So there is no point in choosing K in order to minimize F (you would end up with $K = N$). The best is to look beyond the clustering to the overall aim of the analysis. We will investigate a heuristic method in the examples.

9.1.2. Where K-means fails. These are essentially the cases where the distance used is not a good criterion for clustering. See an example in Figure 9.1.4.

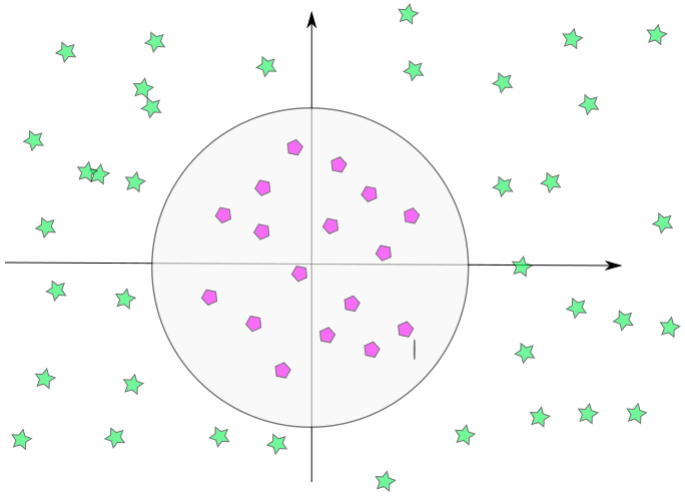


FIGURE 9.1.4. Where K-means fails.

9.1.3. Kernelized K-means. We observe that we only need to compute distances of the kind

$$\begin{aligned} d_{n,k} &= (x_n-\mu_k)^T(x_n-\mu_k) \\ &= \left(x_n-\frac{\sum_{m=1}^N z_k^{(m)}x_m}{N_k}\right)^T\left(x_n-\frac{\sum_{m=1}^N z_k^{(m)}x_m}{N_k}\right) \\ &\quad (\text{where } N_k=\sum_{n=1}^N z_k^{(m)}) \\ &= x_n^Tx_n-\frac{2}{N_k}\sum_{m=1}^N z_k^{(m)}x_n^Tx_m+\frac{1}{N_k^2}\sum_{m=1}^N\sum_{r=1}^N z_k^{(m)}z_k^{(r)}x_m^Tx_n\,. \end{aligned}$$

The idea is to replace the inner products (I call them "scalar products") $x_m^Tx_n$ with a kernel function to get a "kernelized distance". Let us call $Q(\ldots,\ldots)$ this kernel. We get

$$d_{n,k} = Q(x_n,x_n)\frac{2}{N_k}\sum_{m=1}^N z_k^{(m)}Q(x_n^T,x_m)+\frac{1}{N_k^2}\sum_{m=1}^N\sum_{r=1}^N z_k^{(m)}z_k^{(r)}Q(x_m^T,x_n)\,. \tag{9.2}$$

We imagine that $Q(x,y)=\Phi(x)^T\Phi(y)$ for some mysterious transformation Φ which renders the clusters easy to separate. As in the SVM chapter, we do not need to know Φ explicitly. The mean of the k-th cluster becomes

$$\mu_k=\frac{\sum_{n=1}^N z_k^{(n)}\Phi(x_n)}{\sum_{n=1}z_k^{(n)}}\,. \tag{9.3}$$

We modify the algorithm i the following way. We start by randomly initializing the $z_k^{(n)}$ (under the constrains: $z_k^{(n)}\in\{0,1\}$, $\sum_{k=1}^K z_k^{(n)}=1$, for all n).

(1) Compute $d_{n,1},\ldots,d_{n,K}$ for each x_n (using Equation (9.2) above).
(2) Assignment/expectation step. Assign each object (/point) to the cluster with the lowest $d_{n,k}$:

$$z_{n,k}=\begin{cases}1 & \text{if } d_{n,k}=\min_i d_{n,i}\,,\\ 0 & \text{otherwise }.\end{cases}$$

(3) Ghost step (for the update/maximization). If we want to be as close as possible to the algorithm in the previous section, we need to update the (μ_k)'s using Equation (9.3). Obviously, we cannot do this if we do not have access to Φ. So, we do not do it (we imagine it is done in theory).
(4) If assignments have changed in Step 2, return to Step 1, otherwise STOP.

We observe that, in Step 1, we do not need the (μ_k)'s. This is why it is OK that Step 3 is done only in our head.

REMARK 9.1. The K-means algorithm is sensitive to initialization so we better choose the $z_k^{(n)}$ in a non-random way (look at the data, perform a standard K-means, ...).

REMARK 9.2. Some kernels are designed to work well on certain types of objects (texts, graphs, networks ...).

9.1.4. Advantages/disadvantages of K-means methods.
Advantages.

- Relatively simple to implement.
- Scales to large data sets.
- Generalizes to clusters of different shapes and sizes, such as elliptical clusters.
- Always yields a result (also a con, as it may be deceiving).

Disadvantages.

- Choosing K manually.
- Being dependent on initial values. You might have to try many times with different initial values.

- Scaling with number of dimensions. As the number of dimensions increases, a distance-based similarity measure converges to a constant value between any given examples. You should try to reduce the number of dimensions (for example, use PCA).
- Sensitive to outliers. Remove outliers before proceeding with the algorithm.

9.2. Hierarchical clustering

There are two types of hierarchical clustering: agglomerate clustering and divisive clustering, we will not talk about divisive clustering for lack of time.

The algorithm begins with every point representing a single cluster (N points at the beginning). At each step (total of $N-1$ steps), the closest two clusters are merged into a single cluster. To explain what we mean by closest clusters, we define a measure of dissimilarity.

We fix a distance $d(\ldots,\ldots)$ in our data space. For point x_i, x_j, we set $d_{i,j} = d(x_i, x_j)$ and we call this the pairwise observation dissimilarity. When we have two clusters G and H, we can compute the dissimilarity $d(G,H)$ in different ways.

- Single linkage (SL): $d_{SL}(G,H) = \min_{i\in H, j\in G} d_{i,j}$ (also called the the nearest neighbor technique).
- Complete linkage: $d_{CL}(G,H) = \max_{i\in G, j\in H} d_{i,j}$.
- Group average (GA): $d_{GA}(G,H) = \frac{1}{\#G\times\#H}\sum_{i\in G}\sum_{j\in H} d_{i,j}$.

If clusters are compact, the three methods will produce similar results.

- SL can produce clusters where observations are linked by a series of close intermediate observations (this referred to as chaining). See Figure9.2.1.

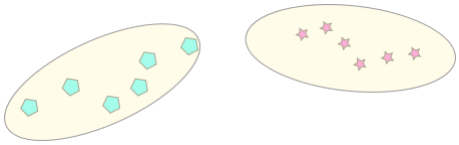


FIGURE 9.2.1. SL

- CL produces compact clusters with small diameters. It can produce clusters such that points in a cluster are much closer to members of other clusters than they are to some members of their own cluster. See Figure 9.2.2.

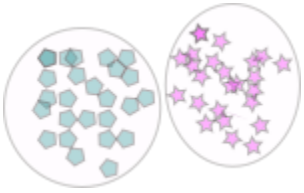


FIGURE 9.2.2. CL

- GA represents a compromise between the two extremes. If you apply a monotone strictly increasing transformation h to the $d_{i,j}$ ($h_{i,j} = h(d_{i,j})$) can change the result (it would not change the result for CL and SL).

Statistical consistency of the GA method. If the population in cluster G is distributed according to some law p_G, and the population in cluster H is distributed according to some law , then $d_{GA}(G,H)$ is a Monte-Carlo estimate of

$$\int\int d(x,x')p_G(x)p_H(x')dxdx' \tag{9.1}$$

(estimate is better when $\#G$ and $\#H$ go to infinity. For the other methods, it is likely that

$$\begin{cases} d_{SL}(G,H) \to 0, \\ d_{CL}(G,H) \to \infty. \end{cases}$$

Binary trees

In all cases, the dissimilarity between merged clusters is monotone, increasing with the level of the merger (see Figure 9.2.3 for an example where we measure the distances between clusters with d_{SL}). Thus the binary tree representing the clustering process can be plotted so that the height of each node

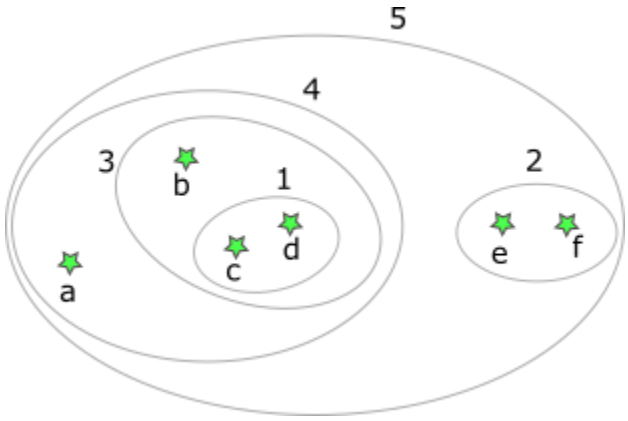


FIGURE 9.2.3. Clusters obtained by the Single Linkage method

is proportional to the value of the intergroup dissimilarity between its two daughters (see Figure 9.2.4). This type of graphical display is called a dendogram.

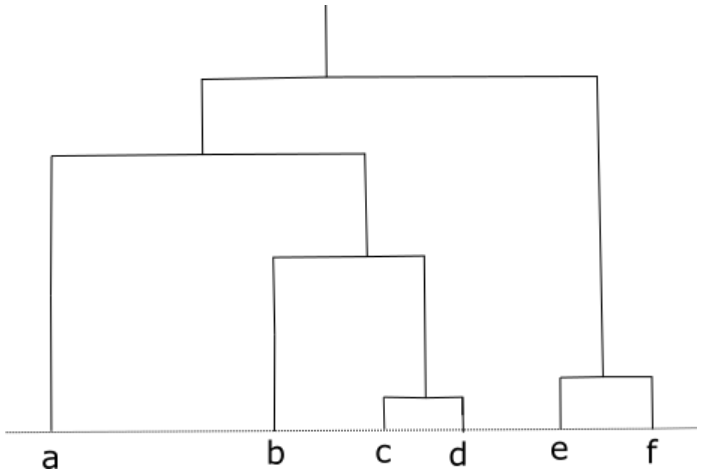


FIGURE 9.2.4. Dendogram

9.2.1. Advantages/disadvantages of agglomerative clustering.

Advantages.

- Single-link algorithms are best for capturing clusters of different sizes and shapes.
- Complete link and group average are not affected by noise.
- Good when data has an underlying hierarchical structure, such as correlations in financial markets (it is also a con).

Disadvantages.

- Single-link algorithms are sensitive to noise.
- Time complexity: not suitable for large datasets.
- Very sensitive to outliers.

9.3. Density-based spatial clustering of applications with noise (DBSCAN)

9.3.1. Definitions. We have a set of points we want to cluster. We have to specify two parameters: $\epsilon > 0$, $m \in \mathbb{N}^*$ (minimum number of points).

DEFINITION 9.3. A point p is a core point if at least m points are within distance ϵ of it (including p). In Figure 9.3.1, we see that A is a core point.

DEFINITION 9.4. A point q is directly reachable from p if q is within distance ϵ from point p.

DEFINITION 9.5. A point q is reachable from core point p if there is a path $p_1, \ldots, p_n$ with $p_1 = p, \ldots, p_n = q$ and each p_{i+1} is directly reachable from p_i. In Figure 9.3.1, we see that the point on the right is reachable from A.

DEFINITION 9.6. All points which are non-reachable from any other points are outliers. In Figure 9.3.1, we that the point on top of the picture is not reachable.

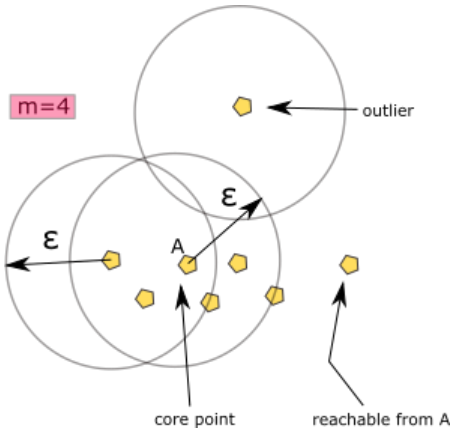


FIGURE 9.3.1. DBSCAN

DEFINITION 9.7. If p is a core point, it forms a cluster with all points that are reachable from it. In a cluster, points which are non-core points form the edge of the cluster.

DEFINITION 9.8. Two points p and q are density-connected if there is a point o such that p and q are reachable from o

A cluster then satisfies two properties.

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

9.3.2. Algorithm.

(1) Find the points in the ϵ-neighborhood of every point, and identify the core points (those with at least m points in their ϵ-neighborhood).
(2) Find the connected components of core points in the (ϵ)-neighbor graph (ignoring all non-core points).
(3) Assign each non-core point to a nearby cluster or assign it to noise (when it is an outlier).

We see a result of this algorithm in Figure 9.3.2. The core points (in solid green, solid blue or solid burgundy) are connected to points of the same color by ϵ-paths (distance between two components is less than ϵ) going only through core points. These core points form the core of the clusters. The points which are not core points are assigned to a cluster if they are reachable from this cluster. Thus, we see that some points are colored in green, blue, burgundy. Observe that, at least for one point which could be green or burgundy, the assignment is arbitrary (and thus a little bit random, because it depends on how the algorithm is coded). Some points cannot be assigned to clusters (the yellow triangles), they are called outliers.

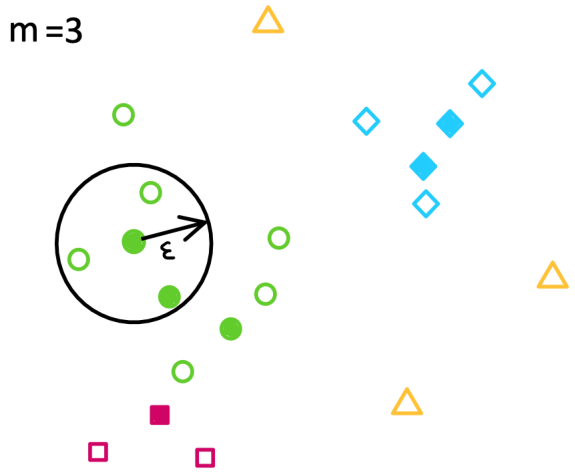


FIGURE 9.3.2. DBSCAN result

9.3.3. Advantages/disadvantages of DBSCAN.
Advantages.

- DBSCAN does not require to specify the number of clusters (contrary to K-means)
- It can find arbitrarily shaped clusters.
- It has a notion of noise so it is robust to outliers.
- DBSCAN requires only two parameters and is mostly insensitive to the points ordering (except for points on the edge).
- The parameters ϵ, m can be chosen by an expert if the data is well understood.

Disadvantages.

- The algorithm is not entirely deterministic (for edge points).
- DBSCAN depends on the distance chosen (euclidean distance will perform poorly in high dimension).
- The parameters ϵ, m can be difficult to choose.

Rule of thumb for the parameters: $m = \max(2 \times D, 3)$ (where D is the dimension).

9.4. Examples in R

9.4.1. Mall Customer Segmentation. We download the data[1] (file `Mall_Customers.csv`). You are owing a supermarket mall and through membership cards, you have some basic data about your customers like Customer ID, age, gender, annual income and spending score. Spending Score is something you assign to the customer based on your defined parameters like customer behavior and purchasing data. You want to understand the customers behavior and segment them into groups that can be targeted by the marketing department.

```
data<-read.csv('datasets/Mall_Customers.csv')
head(data)
```

CustomerID	Gender	Age	Annual.Income..k..	Spending.Score..1.100.
1	Male	19	15	39
2	Male	21	15	81
3	Female	20	16	6

[1] https://www.kaggle.com/vjchoudhary7/customer-segmentation-tutorial-in-python

4	4	Female	23	16	77
5	5	Female	31	17	40
6	6	Female	22	17	76

We transform the gender into a quantitative variable (0 for "Male", 1 for "Female").

```
data$Gender <- ifelse(data$Gender=='Male',0,1)
```

We load the libraries we need.

```
library(stats)
library(factoextra)
library(cluster)
library(dplyr)
```

We remove the customers ID's.

```
customers <- subset(data,-1)
```

As we are going to use measures of a distance that is isotropic, we need to re-scale our data.

```
customers <- scale(customers)
```

K-means try. We want to perform a K-means. In order to select K, we use what is called the elbow method. For various values of K, we plot K vs the sum of squared errors. Let us explain what is the sum of squared error (SSE). We sum within each cluster C with center m:

$$\sum_{j\in C}(x_j - m)^2$$

and then sum the sums. We look for the point representing the "elbow point" (the point after which the line starts decreasing in a linear fashion). The idea is that when we have reached the optimal number of clusters, the sum of squared errors will decrease slower because we break clusters into smaller clusters. See an example in Figure 9.4.1: the SSE is likely to decrease a lot between $K = 1$ and $K = 2$, and to decrease less dramatically afterwards. We look at the graph of K vs the SSE.

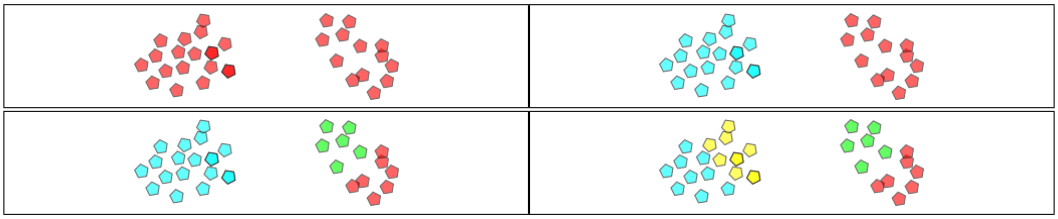


FIGURE 9.4.1. Various K in K-means

```
fviz_nbclust(customers,kmeans,method='wss')
```

We get Figure 9.4.2 We decide to carry on with $K = 6$ (because the graph seems to level after 6) and see if we get meaningful clusters. We perform the K-means:

```
out <- kmeans(customers,iter.max=20,centers=6,nstart=20)
out
```

where

- `iter.max` is the maximum number of iterations
- `centers=...` specifies K
- `nstart` is the number of random sets we try for the initialization (the algorithm keeps the result with the smallest SSE)

We can visualize the clusters using

```
fviz_cluster(out,data=customers)
```

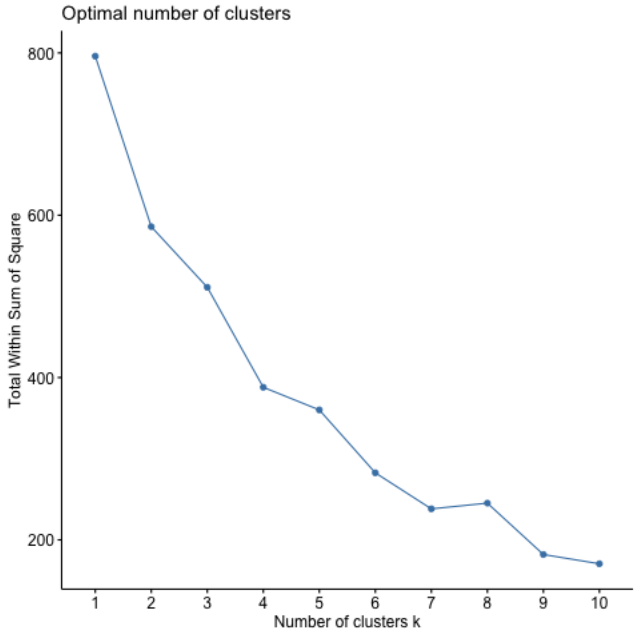


FIGURE 9.4.2. Looking for an elbow

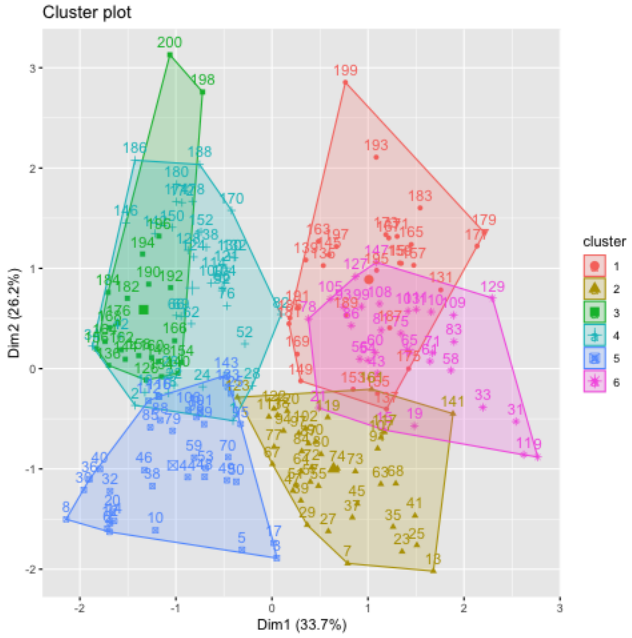


FIGURE 9.4.3. Customers' clusters

(see Figure 9.4.3). Remember the dimension is equal to four (age, annual income, spending score, gender) and we project on the "most meaningful 2D plane using PCA. Let us have a closer look at our clusters. We remove the qualitative `$Gender` column:

```
mall_segmentation<-data[,-1]
```

and add the clusters numbers

```
mall_segmentation$cluster <- out$cluster
```

to the data. We then group elements in the data-frame according to their cluster number and compute the means inside these groups:

```
library(dplyr)
mall_segmentation %>%
        group_by(cluster) %>%
        summarise_all('mean')
```

(remember the use of the pipe operator `%>%` requires the `dplyr` library). We get

```
cluster Gender    Age Annual.Income..k.. Spending.Score..1.100.
*   <int>  <dbl> <dbl>              <dbl>                  <dbl>
1       1  0.433  39.9               90.5                   16.1
2       2  1      50.6               49.7                   40.1
3       3  0.913  32.1               90                     81.4
4       4  0      28.1               58.3                   71.3
5       5  1      25.9               42.2                   57.5
6       6  0      55.9               48.8                   38.8
```

Which means we have the following clusters:

(1) Mixed-gender group, middle-aged, high income, really low-spender.
(2) Essentially women, middle-aged to fifty, average income, medium-spender.
(3) Essentially women, in their thirties or younger, high income, high-spender.
(4) Essentially men, young (in their twenties), medium income, high-spender.
(5) Essentially women, young (in their twenties), medium to low income, medium-spender.
(6) Essentially men, aged, medium income, medium-spender.

Does this make meaningful cluster? To answer this question, you will have to debate with your marketing department.

Kernelized K-means try. We now want to try a kernelized K-means. We have first to make a matrix out of our data.

```
custMat <- as.matrix(customers)
```

Then we use the `kkmeans` command

```
library(stats)
kkout <- kkmeans(custMat,centers=6,kernel='rbfdot',kpar=list(sigma=1))
```

(here we specify we want $K = 6$, a Gaussian kernel with parameter $\sigma = 1$). But we can also let `R` choose the best σ in a mysterious way:

```
kkout <- kkmeans(custMat,centers=6)
```

(the cluster numbers are stored in `kkout@.Data`)

As before, we can look at our clusters.

```
mall_segmentation<-data[,-1]
mall_segmentation$cluster <- kkout@.Data
library(dplyr)
mall_segmentation %>%
        group_by(cluster) %>%
        summarise_all('mean')
```

```
  cluster Gender    Age Annual.Income..k.. Spending.Score..1.100.
*   <int>  <dbl> <dbl>              <dbl>                  <dbl>
1       1  1      26.5               38.9                   57.1
2       2  0      28.8               62.0                   73.2
3       3  0.0270 55.1               53.4                   35.8
4       4  1      32.2               86.8                   82.2
```

5	5 0	30	85.1	14.6
6	6 1	47.6	62.8	36.9

Which means we have the following clusters

(1) Essentially women, young (in their twenties), low income, medium-spender (cluster 5 in the above Section?).
(2) Essentially men, young (in their twenties), medium income, high-spender (cluster 4 in the above Section?).
(3) Essentially men, in their fifties, medium income, low spender (cluster 6 in the above Section?).
(4) Essentially women, in their thirties or younger, high income, high-spender (definitely cluster 3 in the above Section).
(5) Essentially women, in their thirties or younger, high income, low-spender.
(6) Essentially women, middle-aged, average to high-income, medium to low spender.

Again, these clusters have to be discussed with your marketing department.

DBSCAN try. We want to try a DBSCAN where the minimal number of points is $m = 4$. In order to find the best ϵ, we use the following rule of thumb. We calculate the average of the distances of every point to its m nearest neighbors. Next, these m-distances are plotted in an ascending order. The aim is to determine the "knee" which corresponds to the optimal ϵ parameter. We use

```
library(dbscan)
kNNdistplot(custMat, k=4)
abline(h=0.75, col='red')
```

(see the result in Figure 9.4.4). We draw a horizontal red line at level 0.75. A "knee" corresponds to a

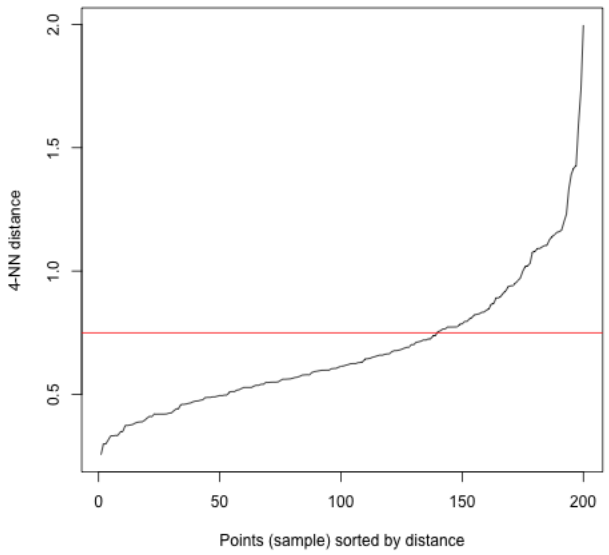


FIGURE 9.4.4. Looking for a knee

threshold where a sharp change occurs along the m-distance curve. The idea is that, for a lot of points, the m-nearest neighbors are at distance 0.75 (approximatively). So we should get nice clusters if we go for a DBSCAN with $\epsilon = 0.75$. In real-life, you would have to try many different m's.

Then we perform a DBSCAN

```
db <- dbscan(customers, eps=0.75, minPts=4)
hullplot(customers, db$cluster)
```

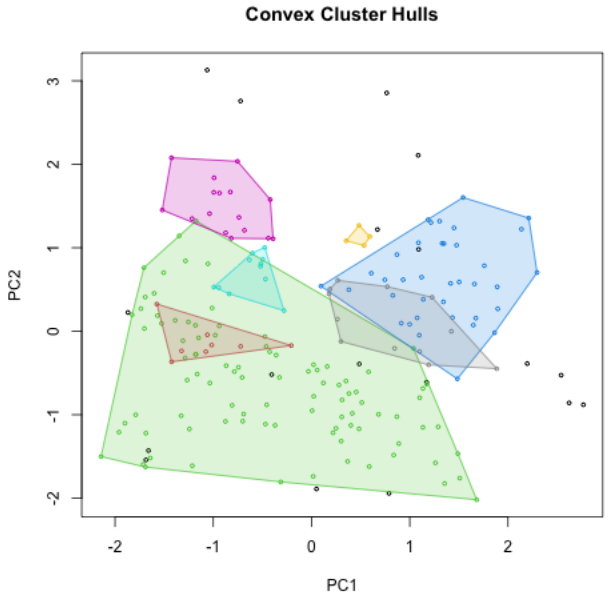


FIGURE 9.4.5. Clusters

and plot the clusters (on the plane determined by PCA), see Figure 9.4.5. Those are quite different from the ones obtained by K-means. We see this as an opportunity to uncover new groups of customers. Observe that the points in black are outliers.

We now want to see the clusters and their cardinality (using similar commands as before)

```
mall_segmentation$cluster <- db$cluster
out <- mall_segmentation %>%
                group_by(cluster) %>%
                summarise_all('mean')
for (i in 0:7)
{
        out$card[i] <- sum(mall_segmentation$cluster == i)
}
out
```

and we get

cluster	Gender	Age	Annual.Income..k..	Spending.Score..1.100.	card
1	0	0.333	38.7	57.1	35.6	8
2	1	0	25.9	24.9	75.9	94
3	2	1	37.5	55.9	56.1	37
4	3	0	52.6	63.5	35.8	11
5	4	0	22.6	55.5	53.3	16
6	5	0	33.6	81.6	83.2	4
7	6	0	20.8	76.2	8	12
8	7	1	43.4	88.3	20.4	24

We observe that some of the clusters were already present in the previous analyses:

(1) Essentially men, young (in their twenties), medium income, high-spender.
(7) Essentially women, middle-aged, average to high-income,medium to low spender.

9.4.2. Cheese classification (hierarchical clustering). We are interested in the classification of a cheese data-set[2]. We download the data

```
fromage <-read.table(file="datasets/fromage.txt",
        header=T,row.names=1,sep="\t",dec=".")
```

(`header=T` means the columns' names are at the beginning of the file, `row.names=1` means the rows' names are in the first column, `sep=''\t''` means the data is separated by tabs, `dec=''.''` means the decimal symbol is the point). We re-scale the data

```
fromage.cr <- scale(fromage)
```

We create the matrix of the distances between individuals

```
library(stats)
d.fromage<-dist(fromage.cr)
```

We perform our hierarchical clustering using the GA (group average) method,

```
cah <- hclust(d.fromage,method="average")
plot(cah)
```

see the plot in Figure 9.4.6. We observe there are fours groups (which we already knew): "fromages

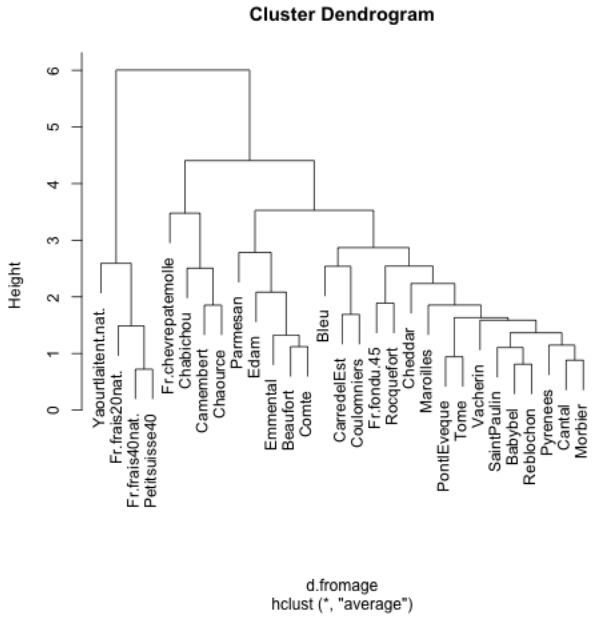


FIGURE 9.4.6. Cheese dendogram

frais" on the left (milk which did not undergo a lot of transformation), then "fromages à pâte molle" (soft cheese), "fromages à pâte dure" (hard cheese), then ... miscellaneous cheeses.

We can materialize the groups on the graph

```
rect.hclust(cah,k=4)
```

(see Figure 9.4.7) and print the groups

```
groupes <- cutree(cah,k=4)
print(sort(groupes))
```

[2] `http://www.math.u-bordeaux.fr/~mchave100p/wordpress/wp-content/uploads/2013/10/fromage.txt`

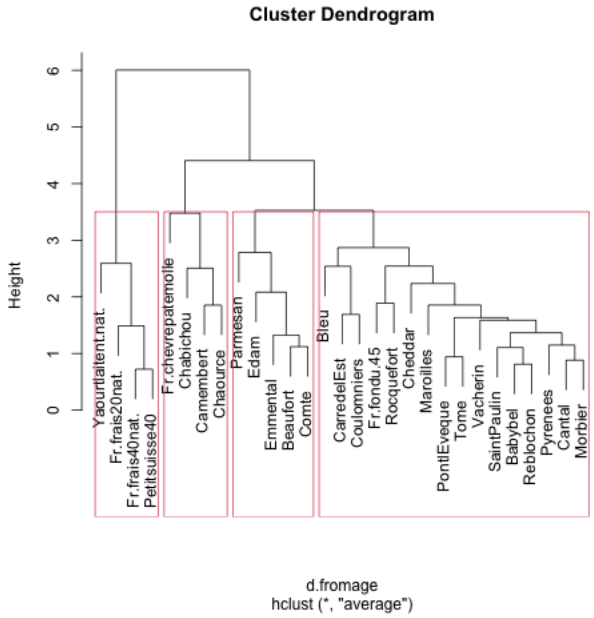


FIGURE 9.4.7. Cheese groups

```
CarredelEst          Babybel             Bleu
          1                1                1
Cantal               Cheddar       Coulomniers
          1                1                1
Fr.fondu.45          Maroilles          Morbier
          1                1                1
PontlEveque          Pyrenees         Reblochon
          1                1                1
Rocquefort           SaintPaulin           Tome
          1                1                1
Vacherin             Beaufort             Comte
          1                2                2
Edam              Emmental             Parmesan
          2                2                2
Camembert            Chabichou          Chaource
          3                3                3
Fr.chevrepatemolle   Fr.frais20nat.     Fr.frais40nat.
          3                4                4
Petitsuisse40     Yaourtlaitent.nat.
          4                4
```

9.4.3. Seeds dataset (activity) ([CNK$^+$10]). We are interested in the dataset[3] `seeds_dataset.txt`. This dataset consists of measurements of geometrical properties of kernels belonging to three different varieties of wheat: Kama, Rosa and Canadian. It has variables which describe the properties of seeds like area, perimeter, asymmetry coefficient etc. There are 70 observations for each variety of wheat. Hint: the file does not have any headers and is tab-separated.

[3] `https://archive.ics.uci.edu/ml/datasets/seeds`

(1) Load the data and name the columns:

```
c('area','perimeter','compactness','length.of.kernel','width.of.kernal',
'asymmetry.coefficient','length.of.kernel.groove','type.of.seed')
```

(2) Check for missing values. Removes lines with missing values.
(3) Hierarchical clustering
   (a) Draw a dendrogram of the data.
   (b) Cut the dendrogram into three clusters.
   (c) Count how many observations were assigned to each cluster.
   (d) Cross-check your clustering results (using `table`).
(4) K-means
   (a) Find the best K with the elbow method.
   (b) Perform K-means and draw the clusters.
(5) DBSCAN
   (a) Find the best ϵ (for $m = 4$) using the knee method.
   (b) Try to find a set of parameters for which we recover three big cluster (hint: that does not seem possible).

9.4.4. Useful commands in R.

```
library(factoextra)
fviz_nbclust(data,kmeans,method='wss')
```

gives us the SSE error as a function of K in K-means (`method='wss'` stands for "plot the SSE").

```
library(stats)
out <- kmeans(customers,iter.max=20,centers=6,nstart=20)
```

performs a K-means, where

- `customer` is the data
- `iter.max` is the maximum number of iterations
- `centers=...` specifies K
- `nstart` is the number of random sets we try for the initialization (the algorithm keeps the result with the smallest SSE)

From there, we can have a nice plot of the clusters using:

```
library(factoextra)
fviz_cluster(out,data=customers)
```

We can group elements in a data-frame by a certain variable using `group_by`. After this, we can compute averages inside the groups using `summarise_all('mean')`.

For kernelized K-means, we use (where `custMat` has to be a matrix)

```
library(kernlab)
kkout <- kkmeans(custMat,centers=6,kernel='rbfdot',kpar=list(sigma=1))
```

(here we specify we want $K = 6$, a Gaussian kernel with parameter $\sigma = 1$). But we can also let R choose the best σ in a mysterious way:

```
kkout <- kkmeans(custMat,centers=6)
```

(the cluster numbers are stored in `kkout@.Data`)

Load a .txt file into R:

```
fromage <-read.table(file="datasets/fromage.txt",
        header=T,row.names=1,sep="\t",dec=".")
```

(`header=T` means the columns' names are at the beginning of the file, `row.names=1` means the rows' names are in the first column, `sep=''\t''` means the data is separated by tabs, `dec=''.''` means the decimal symbol is the point)

Create the matrix of distances between individual from a data-frame (`library(stats)`)

```
d.fromage<-dist(fromage.cr)
```

Hierarchical clustering + plot of the dendrogram (input has to be a distance matrix)

```
cah <- hclust(d.fromage,method="average")
plot(cah)
```

visualize groups on the dendrogram

```
rect.hclust(cah,k=4)
```

and print the groups

```
groupes <- cutree(cah,k=4)
print(sort(groupes))
```

The command

```
kNNdistplot(custMat,k=4)
```

draws the mean distance to the 4 nearest neighbors for all points (these distances are ordered) (the argument `custMat` has to be a distance matrix).

DBSCAN:

```
db <- dbscan(customers,eps=0.75,minPts=4)
```

(`eps` stands for ϵ and `minPts` for m). View the clusters obtained by DBSCAN (on the "PCA plane"):

```
hullplot(customers,db$cluster)
```

CHAPTER 10

Appendix

10.1. Lagrange multipliers

10.1.1. Notations. When $f:\mathbb{R}^n\to\mathbb{R}$ is sufficiently differentiable. We can write a first order development, using many equivalent notations for the first order differential of f:

$$\begin{aligned} f(x+u) &= f(x)+df(x)(u)+o(\|u\|) \\ &= f(x)+f'(x).u+o(\|u\|)\,. \end{aligned}$$

We have

$$f'(x).u=\sum_{i=1}^{n}\frac{\partial f}{\partial x_i}(x)\times u_i\,.$$

So, if we define the gradient of f by

$$\nabla f(x)=\begin{pmatrix}\frac{\partial f}{\partial x_1}(x)\\ \vdots \\ \frac{\partial f}{\partial x_n}(x)\end{pmatrix},$$

we then get

$$f'(x).u=\langle\nabla f(x),u\rangle$$

($\langle\ldots,\ldots\rangle$ is the scalar product).

⚠This is not a real proof.

10.1.2. Optimization under equality constrains. Suppose, we have $f:\mathbb{R}^n\to\mathbb{R}$ and $g_1,\ldots,g_p:\mathbb{R}^n\to\mathbb{R}$. We set $M=\{x:g_i(x)=c_i\,,\,1\le i\le p\}$ (for some constants c_i). We are interested in

$$\operatorname*{arg\,min}_{x\in M} f(x)\,,$$

which is the same as finding

$$\begin{cases}\min f(x)\\ \text{under } g_i(x)=c_i\,,\,1\le i\le p\,.\end{cases}\tag{10.1}$$

Necessary condition. Suppose we have found a maximum in x_0. Then, for any "small" move u such that $g_i'(x_0).u=0$ (that is, a move that stays in M), we have

$$f(x_0+u)=f(x_0)+f'(x_0).u+o(\|u\|)\,.$$

So $f'(x_0).u=0$. This means that

$$\nabla f(x_0)\in\operatorname{Span}(\nabla g_1(x_0),\ldots,\nabla g_p(x_0))\,.$$

Lagrangian function. We set

$$L(x,\lambda_1,\ldots,\lambda_p)=f(x)-\sum_{i=1}^{p}\lambda_i(g_i(x)-c_i)$$

($L:\mathbb{R}^n\times\mathbb{R}^p\to\mathbb{R}$). In x_0, there exist $\lambda_1^{(0)},\ldots,\lambda_p^{(0)}$ such that

$$\nabla f(x_0)=\sum_{i=1}^{p}\lambda_i^{(0)}\nabla g_i(x_0)\,.$$

Let us compute

$$\nabla L(x,\lambda_1,\dots,\lambda_p)=\begin{pmatrix}\frac{\partial f}{\partial x_1}(x)-\sum_{i=1}^p\lambda_i\frac{\partial g_i}{\partial x_1}(x)\\ \vdots\\ \frac{\partial f}{\partial x_n}(x)-\sum_{i=1}^p\lambda_i\frac{\partial g_i}{\partial x_n}(x)\\ -(g_1(x)-c_1)\\ \vdots\\ -(g_p(x)-c_p)\end{pmatrix}.$$

We observe that

$$\nabla L(x_0,\lambda_1^{(0)},\dots,\lambda_p^{(0)})=0\,.$$

Working our way back. The conclusion is that, when trying to find the solution of (10.1), a good candidate is x_0 such that there exists $(\lambda_i^{(0)})_{1\le i\le p}$ with $\nabla L(x_0,\lambda_1^{(0)},\dots,\lambda_p^{(0)})=0$.

The coefficients $\lambda_i^{(0)}$ are called "Lagrange multipliers".

10.1.3. Optimization under inequality constrains. Suppose, we have $f:\mathbb{R}^n\to\mathbb{R}$ and $g_1,\dots,g_p:\mathbb{R}^n\to\mathbb{R}$. We are interested in

$$\begin{cases}\min f(x)\\ \text{under } g_i(x)\le c_i\,,\ 1\le i\le p\,.\end{cases}\tag{10.2}$$

for some $c_1,\dots,c_p$.

Necessary condition. Suppose we have found a maximum x_0. We then divide the indexes into

- for $1\le i\le k$, $g_i(x_0)=c_i$ (we say the constraint is binding)
- for $k+1\le i\le p$, $g_i(x_0)<c_i$ (we say the constraint is not binding)

for some k in $\{0,1,\dots p\}$.

For v such that $g_i'(x_0).v=0$ $(1\le i\le k)$, we have

$$f'(x_0).v=0$$

(because x_0 is a maximum of f restrained to $\{x:g_i(x)=c_i\,,\,1\le i\le k\}$). So there exist $\lambda_1^{(0)},\dots,\lambda_k^{(0)}$ such that

$$\nabla f(x_0)=\sum_{i=1}^k\lambda_i^{(0)}\nabla g_i(x_0)\,.$$

We set

$$\lambda_{k+1}^{(0)}=\dots=\lambda_p^{(0)}=0\,.$$

Let us now take $h>0$ ("small"). We have, for all i in $\{1,2,\dots,k\}$,

$$f(x_0-h\nabla g_i(x_0))=f(x_0)-h\langle\nabla f(x_0),\nabla g_i(x_0)\rangle+o(h)\,,$$

and we should have $f(x_0-h\nabla g_i(x_0))\ge f(x_0)$. So

$$\langle\nabla f(x_0),\nabla g_i(x_0)\rangle\le 0\,.$$

In the case where the $\nabla g_i(x_0)$ are orthogonal, the above implies $\lambda_i^{(0)}\ge 0$.

Lagrangian function. We define

$$L(x,\lambda_1,\dots,\lambda_p)=f(x)-\sum_{i=1}^p\lambda_i(g_i(x)-c_i)$$

($L:\mathbb{R}^n\times\mathbb{R}^p\to\mathbb{R}$). We study L under the constrain: $\lambda_i\ge 0$ $(\forall i)$. The gradient of L is (the same as before)

$$\nabla L(x,\lambda_1,\dots,\lambda_p)=\begin{pmatrix}\frac{\partial f}{\partial x_1}(x)-\sum_{i=1}^p\lambda_i\frac{\partial g_i}{\partial x_1}(x)\\ \vdots\\ \frac{\partial f}{\partial x_n}(x)-\sum_{i=1}^p\lambda_i\frac{\partial g_i}{\partial x_n}(x)\\ -(g_1(x)-c_1)\\ \vdots\\ -(g_p(x)-c_p)\end{pmatrix}.$$

Suppose we find a maximum of F in $(x_0, \lambda^{(0)})$. We get that

$$\begin{cases} \frac{\partial L}{\partial x_1}(x_0, \lambda^{(0)}) = 0, \dots, \frac{\partial L}{\partial x_n}(x_0, \lambda^{(0)}) = 0, \\ \lambda_i^{(0)}(g_i(x_i) - c_i) = 0\,,\ 1 \leq i \leq p\,, \\ g_i(x_0) \leq c_i\,,\ 1 \leq i \leq p\,, \\ \langle \nabla f(x_0), \nabla g_i(x_0) \rangle \leq 0\,,\ 1 \leq i \leq k\,. \end{cases} \tag{10.3}$$

The second line above comes from the fact that if this is not the case, we are not at a minimum (we could find other λ's and x such that the function is smaller).

Working our way back. The conclusion is that, when trying to find the solution of (10.2), a good candidate is x_0 such that there exists $(\lambda_i^{(0)})_{1 \leq i \leq p}$ such that Equation (10.3) is satisfied.

Bibliography

[AH20] Jim Albert and Jingchen Hu, *Probability and bayesian modeling*, CRC Press/Taylor & Francis Group, Bocan Raton, FL, 2020. (document), 8.5.3

[Bin01] Kenneth George Binmore, *Calculus concepts and methods*, Cambridge University Press, Cambridge, 2001. 4.2.2

[BS00] I. Beichl and F. Sullivan, *The Metropolis algorithm*, Computing in Science Engineering **2** (2000), no. 1, 65–69. 8

[CNK+10] Małgorzata Charytanowicz, Jerzy Niewczas, Piotr Kulczycki, Piotr A. Kowalski, Szymon Łukasik, and Sławomir Żak, *Complete gradient clustering algorithm for features analysis of x-ray images*, Information Technologies in Biomedicine (Berlin, Heidelberg) (Ewa PiÈ©tka and Jacek Kawa, eds.), Springer Berlin Heidelberg, 2010, pp. 15–24. (document), 9.4.3

[Cyb89] G. Cybenko, *Approximation by superpositions of a sigmoidal function*, Math. Control Signals Systems **2** (1989), no. 4, 303–314. MR 1015670 6.1.1

[GWHT17] James Gareth, Daniel Witten, Trevor Hastie, and Robert Tibshirani, *An introduction to statistical learning with applications in R*, corrected at 8th printing 2017 ed., Springer texts in statistics, #0, Springer, Springer Science+Business Media, New York, 2017. 7, 7.2.1

[Has95] M. Hassoun, *Fundamentals of artificial neural networks MIT Press*, MIT, 1995. 6.1.1

[Hay98] Simon Haykin, *Neural networks: A comprehensive foundation*, 2nd ed., Prentice Hall PTR, USA, 1998. 6.1.1

[Hor91] Kurt Hornik, *Approximation capabilities of multilayer feedforward networks*, Neural Networks **4** (1991), no. 2, 251 – 257. 6.1.1

[HTF09] Trevor Hastie, Robert Tibshirani, and Jerome Friedman, *The elements of statistical learning*, second ed., Springer Series in Statistics, Springer, New York, 2009, Data mining, inference, and prediction. MR 2722294

[JPR94] Eric Jacquier, Nicholas G. Polson, and Peter E. Rossi, *Bayesian analysis of stochastic volatility models*, Journal of Business & Economic Statistics **12** (1994), no. 4, 371–389.

[KSC98] Sangjoon Kim, Neil Shephard, and Siddhartha Chib, *Stochastic volatility: Likelihood inference and comparison with arch models*, The Review of Economic Studies **65** (1998), no. 3, 361–393.

[MMG18] A. MULLER, A.C. MU+00fcller, and S. Guido, *Introduction to machine learning with python: A guide for data scientists*, O'Reilly Media, Incorporated, 2018. 1.4.7

[MP44] Warren S. McCulloch and Walter Pitts., *A logical calculus of the ideas immanent in nervous activity. bulletin of mathematical biophysics, vol. 5 (1943), pp. 115?133.*, Journal of Symbolic Logic **9** (1944), no. 2, 49?50. 6.1.2

[Mur13] Kevin P. Murphy, *Machine learning a probabilistic perspective*, Just the fact101 study guide, #0, Cram101, [Moorpark, California], 2013.

[Par08] Etienne Pardoux, *Markov processes and applications*, French ed., Wiley Series in Probability and Statistics, John Wiley & Sons, Ltd., Chichester; Dunod, Paris, 2008, Algorithms, networks, genome and finance. MR 2488952 (2010g:60001)

[PYLS20] Sejun Park, Chulhee Yun, Jaeho Lee, and Jinwoo Shin, *Minimum width for universal approximation*, 2020. 6.1.1

[RG17] Simon Rogers and Mark Girolami, *A first course in machine learning*, 2nd edition ed., Chapman & Hall/CRC machine learning & pattern recognition series, #0, CRC Press, Boca Raton, 2017.

[Rip96] B. D. Ripley, *Pattern recognition and neural networks*, Cambridge University Press, Cambridge, 1996. MR 1438788 7.1.1

[WRDV+17] Tong Wang, Cynthia Rudin, Finale Doshi-Velez, Yimin Liu, Erica Klampfl, and Perry MacNeille, *A bayesian framework for learning rule sets for interpretable classification*, Journal of Machine Learning Research **18** (2017), no. 70, 1–37. (document), 7.3.4

List of symbols

$:=$	We define the term of the left by this equality.
$\#$	Cardinality.
$\mathbb{1}$	Indicator function.
δ_x	Dirac measure in x.
$\langle \ldots, \ldots \rangle$	Scalar product.
$\Leftrightarrow$	If and only if.
$\mathscr{B}(p)$	Bernouilli law of parameter p.
$\mathscr{P}(\lambda)$	Poisson distribution with parameter λ.
$\mathscr{U}([a,b])$	Uniform law on $[a,b]$.
∇	Gradient.
$\propto$	Proportional to ...
$\sim$	Equivalent to ...
Span	Vector space spanned by ...
⚠	Danger ahead
$\mathscr{L}$	Law.
💡	Important idea.

Index

A posteriori distribution, 24
A posteriori law, 3
A priori density, 99
A priori distribution, 24
A priori law, 3
Accuracy, 63
Activation function, 66

Back-propagation, 68
Backward pss, 69
Bag-of-words, 55
Bagging, 86
Basis function expansion, 12
Batch-learning, 69
Bayesian statistics, 3, 24
Bernoulli, 32
Binary classification, 2
Boosting, 79
Bootstrap aggregation, 86
Boxes, 80
Bradley-Terry model, 109
Branches, 80
Burn in, 100

CART, 79
Classification error rate, 82
Clusters, 5
Configurations space, 100
Confusion matrix, 17, 63
Consistent estimators, 3
Cost complexity pruning, 82
Critical point, 23
Cross-entropy, 67
Cross-validation, 15
Curse of dimensionality, 11

DBSCAN, 127
Deciphering, 106
Decision boundary, 13, 41
Decision function, 41
Decision tree, 79
Descriptive learning, 2
Dimensionality reduction, 6
Dummy variables, 27, 28, 76, 86

Elbow method, 129
Entropy, 83
Euler Gamma function, 119

Feedback neural network, 66
Feedforward neural network, 66
Forward pass, 69

Gamma distribution, 115
Gamma law, 119
Generalization error, 15
Gibbs sampler, 100, 105
Gini index, 83
Gradient descent, 35
Greedy, 81

Handwriting recognition, 5
Hard margin, 45
Hat matrix, 23
Hidden variable, 6

Improper prior, 115
Instance-based learning, 10
Internal nodes, 80
Iterated re-weighted least square, 37

Kernel functions, 47
Kernelization, 48
KNN, 10

Lagrange multipliers, 43, 139
LASSO, 25, 82
Latent variable, 6
Least squares regression, 21
Leaves, 80
Likelihood, 24, 109, 119
Line minimization, 35
Line search, 35
Linear regression, 12
Linear support vector machine, 41
Linearly separable, 14, 41
Logistic regression, 12
LOOCV, 15

Majority vote, 87
Maximum a posteriori, 24
Maximum Likelihood Estimator, 32
Maximum likelihood estimator, 20
Memory-based learning, 10
Metropolis, 105
Metropolis-Hastings algorithm, 104
MH, 105
Misclassification rate, 15
MLE, 20
Model, 2, 3
Model selection, 15
Multi-class classification, 2

NBC, 56
Negative log-likelihood, 21
Newton's method, 36
NLL, 21

OOB, *see* Out-of-Bag
Out-of-Bag, 87
Over-fitting, 14, 37, 71, 81

Parametric vs. non-parametric, 10
Pareto, 114
Penalization term, 37
Penalized least squares, 25
Penalty, 25
Pipe operator, 60, 76
Poisson distribution, 119
Posterior, 99, 110, 119
Posterior density, 99
Prediction error, 20
Predictive learning, 1
Principal Component Analysis, 6
Prior, 99, 110, 119
Proposal kernel, 104
Purity, 83

Re-scaling, 10, 51
Recall, 32
Recursive binary splitting, 81
Residual sum of squares, 21
Ridge regression, 25
RSS, 21, 80

Sensitivity, 32
Shrinkage methods, 25
Sigmoid function, 12
Soft-max function, 66
Sparse solution, 45
SSE for K-means, 122
Steepest descent, 35
Subtree, 81
Supervised learning, 1, 2
Survival function, 117
SVM, 41

Target advertising, 51
Target law, 104, 110
Target variable, 19
Terminal nodes, 80
Test set, 15
Top-down, 81
Training epoch, 69
Training set, 1
Tree pruning, 81

Under-fitting, 14
Universal approximation theorem, 65
Unsupervised learning, 2

Validation set, 15

Weakest link pruning, 82

Zero probability problem, 58