DEGREE'S FINAL PROJECT

Use of artificial neural networks for emergencies prediction at Hospital Universitari de Girona Dr. Josep Trueta

Eduard Barroso & Sergi Rivas
edu.barrosod@gmail.com; sergirivas.92, @gmail.com

Escola Tècnica Superior d'Enginyeria Industrial de Barcelona

UNIVERSITAT POLITÈCNICA DE CATALUNYA

ETSEIB

1. Abstract

The motivation of the project is to model and predict the volume of arrivals at the emergency department (ED) of a general hospital. The process consists of complex linear and nonlinear patterns together. Those types of temporal series are tough to solve efficiently using Box-Jenkins methods (ARIMA models) due its high stochastic behaviour and nonlinearity.
Once the time series analysis is discarded owing the bad results obtained, and in order to change the approach of the task, artificial neural networks (ANN) are chosen to solve the problem. This methodology offers a whole new perspective of study, enabling the use of algorithms in a high tight time constraint in order to predict intraday information such as the arrivals expected to occur in the afternoon using morning information.
The objective is to program a plain applicative, able to extract the data needed (endogenous and exogenous variables), compute the ANN algorithm and finally show the relevant results in order to help improving the human and material resources management in the area of emergencies. As a fundamental part of the project, the best methodology to work with ANN algorithms is seek in order to settle an accurate approach for future studies.

2. Summary

1. Abstract 2

2. Summary 3

3. Introduction 6

3.1. Purpose of the project 6

3.2. Scope of the project 6

4. Body of the report 7

4.1. Planning 7

4.2. Processing the data 8

4.2.1. Code structure 8

4.2.2. Functions 8

4.2.3. Database creation 13

4.3. Descriptive analysis 15

4.3.1. Preview of the system 15

4.3.2. Characterization of the system 15

4.4. Time series analysis 19

4.4.1. Conclusions for the Time Series analysis 24

4.5. Artificial Neural Networks 25

4.5.1. Neural Net package 25

4.5.1.1. ANN first font code 25

4.5.1.2. ANN first predictions 27

4.5.1.3. Nnet package 30

4.5.2. Nnet package 30

4.5.2.1. Disaggregate the information 32

4.5.2.2. Data Sets 32

4.5.2.3. Respiratory type 33

4.5.2.4. Organic type 35

4.5.2.5. Cardiovascular type 36

4.5.2.6. Features for each data set 36

4.5.2.7. Features in detail 38

4.5.2.7.1 Respiratory type 38

4.5.2.7.2 Organic type 39

4.5.3. R Stuttgart Neural Network Simulator library 40

4.5.3.1. Learning in RSNNS 40

4.5.3.1.1 General learning procedure 40

4.5.3.1.2 Backpropagation learning functions 41

4.5.3.1.3 Quickpropagation learning function 42

4.5.3.1.4 Resilient propagation learning function 42
4.5.3.2. Update Functions in RSNNS 43
4.5.3.3. Initialization Functions in RSNNS 44
4.5.3.4. Algorithm development 44
4.5.3.4.1.1 Variable introduction 44
4.5.3.4.1.2 Error analysis 45
4.5.3.4.1.3 Cross Validation 51
4.5.3.4.2 Consistency analysis 51
4.5.3.4.2.1 Variable normalization 53
4.5.3.5. Topology selection 54
4.5.3.5.1.1 Factorial experiment 54
4.6. Artificial Neural Networks: python implementation 64
4.6.1. Why Python? 64
4.6.2. Steps before training 65
4.6.3. PyBrain 66
4.6.3.1. PyBrain code 66
4.6.4. Class implementation 71
4.6.4.1. ANN code 71
4.6.4.2. Python conclusions 78
5. Conclusions 79
6. Economic study 81
7. Environmental study 84
8. Bibliography 85
9. ANNEX 1 87
9.1. Core of the algorithm 87
9.2. Factorial experiment code 92
9.3. 94
9.4. PyBrain documentation 95
9.4.1. Create and process the data sets 95
9.4.2. Define the layers structure and oolean 96
9.4.3. Build a standard multilayer perceptron 96
9.4.4. Define the training algorithm 96
9.5. Auxiliary python libraries 98
9.5.1. Math 98
9.5.2. Random 98
9.5.3. Numpy 98
9.5.4. Datetime 100
9.5.5. Matplotlib 100

9.6. R documentation....103
9.6.1. Nnet....103
9.6.2. RSNNS....104
9.6.2.1. mlp....104
9.7. Futures paths to follow....105
9.7.1. Support vector regression....105
9.7.2. How it Works?....105
9.7.3. SVR over sklearn....106

ETSEIB

3. Introduction

3.1. Purpose of the project

The goal of the project is to create a useful application for the Hospital Universitari de Girona Dr. Josep Trueta in order to control and predict the volume of arrivals at the emergency department (ED). The whole day (24 hours) has been divided in two particular strips: from 07:00 to 13:00 corresponding to the first strip, and the rest of the day as the second strip.
The executable, using the appropriate covariates (extracted from the first strip) and algorithms programmed (ANN), should be able to predict the amount of patients left to come in the same day for the last strip. By estimating the parameters with good accuracy, the applicative might be useful for the institution in order to improve the resource management and logistics of this particular area as they program the second part of the day after midday.

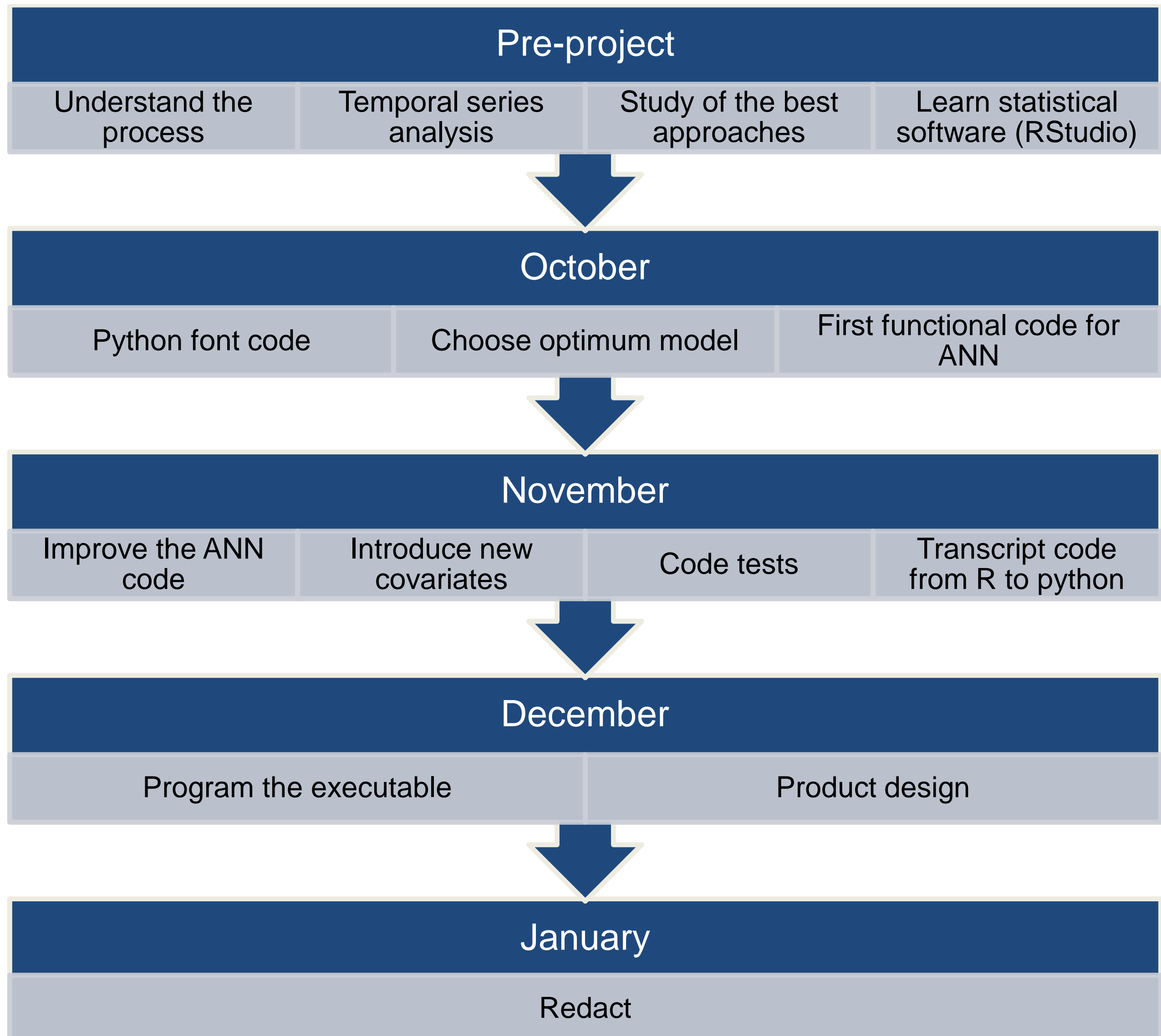
3.2. Scope of the project

In order to ensure the functionality of the executable, there are two important topics that must be accurately solved. First of all, from a full analysis and thanks to a deep understanding of the general flow of the process, select the right and more explicative covariates. And second, choose and develop the proper algorithm to fit the real model in the best possible way.

The first point has been solved by a large and deep descriptive analysis using all the data available in order to understand the process and their explicative variables.
The second has been approached using the time series Box-Jenkins methodology (ARIMA). Despite this is a quite powerful tool for time series analysis, in these particular cases, where the data has a predominant random behaviour, the methodology does not offer the accuracy required. After comparing and studying different options, the knowledge and experience leads to use soft computing tools such as artificial neural networks (ANN) or support vector machine (SVM) algorithms. ANN's offer universal function approximation capability on the data itself, i.e. they are purely empirical models that can theoretically mimic any relation to any nine-degree of precision. In practicality they have limitations in their approximation capability by finite and noisy data sets, and stochastic relationships that can mask the final results.
So, once the main characteristics are analysed this methodology seems enough powerful and appropriate to face the posed problem.

4. Body of the report

4.1. Planning



4.2. Processing the data

To deal with the huge amount of data generated each year by the hospital, some tools like spreadsheets or statistical software are not flexible enough to manipulate the volume of data. For this reason a *python* plug-in is required to process the data and create a compatible format for the *R* scripts.

4.2.1. Code structure

The data brought by the hospital is divided in two files that have to be merged in a single database. To do so, a python code has to parse both files, drop all the information into a python compatible structure and proceed to join the data. Those steps are implemented in separated functions and can be easily adapted to bring different data sets.

The main difficulty of this problem is the time structure. All the data in the files is referenced by a typical date, thus it is necessary to translate it to a single magnitude such as hours. All those calculations are performed with the python library *datetime* taking 01/01/2010 as time reference.

4.2.2. Functions

Given two dates returns the difference between both in years. This function is a key element to reduce the time schema into a single magnitude.

```
from datetime import datetime, date

def time_to_years(reference_date, current_date):
    FMT = '%d/%m/%Y'
    t1 = datetime.strptime(reference_date, FMT)
    t2 = datetime.strptime(current_date, FMT)
    t_difference = t2 - t1
    time = int(t_difference.days/365)
    return time
```

Parses each attribute from a given *icodi* line according to their type (either integer or string). Return the processed line in a list. This function is part of the process of reading the file and is not intended to work by itself.

```
def processIcodiRow(icodiLine):
    icodi = int(icodiLine[0])
    diagnosticCode = icodiLine[1]
    ps = icodiLine[2]
    return [icodi, diagnosticCode, ps]
```

Parses each attribute from a given *exmu* line, it according to their type. In addition it formats the dates and times to fit a standard format. Returns the processed line in a list. This function is part of the process of reading the file and is not intended to work by itself.

```
def processExmuRow(exmuLine):
    exmuCode = int(exmuLine[0])
    dateIn = exmuLine[1][6:8] + "/" + exmuLine[1][4:6] + "/" + exmuLine[1][0:4]
    hourIn = exmuLine[2][:-4] + ":" + exmuLine[2][-4:-2] + ":" + exmuLine[2][-2:]
    dateOut = exmuLine[3][6:8] + "/" + exmuLine[3][4:6] + "/" + exmuLine[3][0:4]
    hourOut = exmuLine[4][:-4] + ":" + exmuLine[4][-4:-2] + ":" + exmuLine[4][-2:]
    rawAge = exmuLine[5][6:8] + "/" + exmuLine[5][4:6] + "/" + exmuLine[5][0:4]
    age = time_to_years(rawAge, dateIn)
    gender = exmuLine[6]
    return [exmuCode, dateIn, hourIn, dateOut, hourOut, age]
```

Parses a given file according to its type (either *icodi* or *exmu*). Returns a list of the data by rows. The *fileType* is a string, which indicates the type of the file being parsed (*icodi* or *exmu).* The isHeader bool indicates if the file being parsed has a header indicating the value in each column. In case this flag is set, the first line of the file is skipped. If any of the lines, outputs an error while parsing, a warning is shown with all the relevant information.

```
def readFile(fileIn, fileType, isHeader = True):
    count = 0
    dataLog = []
    for line in fileIn:
        if isHeader:
            isHeader = False
        else:
            try:
                line = line.split("\r\n")[0]
                line = line.split(";")
                if fileType == "icodi":
                    row = processIcodiRow(line)
                elif fileType == "exmu":
                    row = processExmuRow(line)
                else:
                    row = line
                dataLog.append(row)
            except:
                print "Error parsing line [%d]: %s" % (count, line)
            count += 1
    dataLog.sort(key = lambda x: x[0])
    return dataLog
```

Simplifies the *icodi* data by eliminating the non-principals causes and selecting the given typology. To simplify each patient's data, the *icodi* register is sorted and only the main clinical profile is stored. In case the process fails, the line is skipped.

```
def simplifyIcodiFile(icodiLogAux, patientType):
    codi = 0
    icodiLog = []
    isAdded = False
    isPrincipal = False
    for log in icodiLogAux:
        if log[0] != codi:
            if log[2] == "P":
                isPrincipal = True
                isAdded = True
            else:
                isPrincipal = False
            codi = log[0]
        elif not isPrincipal:
            if log[2] == "P":
                isPrincipal = True
                isAdded = True
        if isAdded:
            try:
                icodi = int(log[1].split(".")[0].split("/")[0])
                if (390 <= icodi) and (icodi <=459):
                    currentType = "car"
                elif (460 <= icodi) and (icodi <=519):
                    currentType = "res"
                else:
                    currentType = "org"
                if currentType == patientType:
                    icodiLog.append(log)
            except:
                isPrincipal = False
            isAdded = False
    return icodiLog
```

Returns the inner join between two given tables and two attributes. This function is very useful when merging the *icodi* and *exmu* registers, and is equivalent to the relational algebra operation.

```
def join(tableA, tableB, colA, colB):
    """
    Inner join statement that uses an equivalence operation (colA = colB)
    to match rows from different tables
    """
    dataLog = []
    idx = 0
    for idxA in range(len(tableA)):
        for idxB in range(idx, len(tableB)):
            rowA = tableA[idxA]
            rowB = tableB[idxB]
            if rowA[colA] == rowB[colB]:
                row = rowA + rowB[0:colB] + rowB[colB + 1:]
                dataLog.append(row)
                idx = idxB
                break
    return dataLog
```

Returns a table with the number of entries per day and hour. To process this information a matrix of size number of hours per number of days is set, and each patient is assigned to its corresponding position. In case any of the steps computing the patient's information fails, an error is raised and a warning indicates the percentage of errors during the process.

```
def flowPerHour(dataBase, referenceDate, endDate):
    count = 0
    errorCount = 0
    totalDays = abs(endDate - referenceDate).days
    flowIn = [ ([0] * 24) for row in xrange(totalDays) ]
    for row in dataBase:
        try:
            dateInRaw = row[3].split("/")
            dayIn = int(dateInRaw[0])
            monthIn = int(dateInRaw[1])
            yearIn = int(dateInRaw[2])
            dateIn = date(yearIn, monthIn, dayIn)
            daysIn = abs(dateIn - referenceDate).days
            hourIn = int(row[4].split(":")[0])
            flowIn[daysIn][hourIn] += 1
        except:
            errorCount += 1
        count += 1
    print 100 - (errorCount*100 / count), "% of the data was correctly parsed"
    return flowIn
```

ETSEIB

Computes the capacity flow within a range of hours in a day. Once the number of arrivals is computed for all the hours and days, the matrix is simplified into two main strips. Those intervals are set by the flags *lowerBound* and *upperBound* and are restricted to:

$$s.t.\begin{cases} 0 \leq \text{lowerBound} < 24 \\ 0 \leq \text{upperBound} < 24 \\ \text{upperBound} \neq \text{lowerBound} \end{cases}$$

```
def computeStrip(dataBase, lowerBound, upperBound):
    """
    Computes the flow per strips in the interval [lowerBound, upperBound)
    """
    if lowerBound < 0 or upperBound > 24 or lowerBound == upperBound:
        print "Invalid range of hours"
        return []

    outputLog = []
    if lowerBound > upperBound:
        for day in dataBase:
            totalPatients = 0
            for hour in range(0, upperBound):
                totalPatients += day[hour]
            for hour in range(lowerBound, 24):
                totalPatients += day[hour]
            outputLog.append(totalPatients)
    else:
        for day in dataBase:
            totalPatients = 0
            for hour in range(lowerBound, upperBound):
                totalPatients += day[hour]
            outputLog.append(totalPatients)
    return outputLog
```

Writes a file given a path and a database. The file is written in *csv* format separating each field of the database with a “;”.

```
def writeFile(outputFile, dataBase):
    for row in dataBase:
        line = str(row[0])
        for atrb in range(1, len(row)):
            line = line + ";" + str(row[atrb])
        outputFile.write(line + "\n")
    return
```

4.2.3. Database creation

All the functions described above, are used to summarize in a database all the information from the years 2010 to 2013. For each year the *icodi* is red, the *exmu* file is red, the *icodi* is simplified, the tables are merged and the information is added to the database.

```
dataBase = []
for year in [2010, 2011, 2012, 2013]:
    icodiFile = open('icodi_' + str(year) + '.csv', 'r')
    exmuFile = open('exmu_' + str(year) + '.csv', 'r')

    icodiLogAux = readFile(icodiFile, "icodi")
    icodiLog = simplifyIcodiFile(icodiLogAux, "res")
    exmuLog = readFile(exmuFile, "exmu")
    dataLog = join(icodiLog, exmuLog, 0, 0)
    dataBase += dataLog
```

The output during the process, show some errors parsing the lines from the *icodi* and *exmu* files. Those results are normal as the data is not completely consistent in its form and range of values and the result is a huge database stored in the variable *dataBase*.

```
Error parsing line [76496]: ['U208046165', '728.85/1', 'P']
Error parsing line [76497]: ['U208046165', '847.0/5', 'P']
Error parsing line [76498]: ['U208060750', '557.1/1', '']
Error parsing line [76499]: ['U208060750', '401.9/2', '']
Error parsing line [76500]: ['U208060750', '427.31/2', '']
Error parsing line [76501]: ['U208060750', '414.9/2', '']
Error parsing line [76502]: ['U208060750', '425.4/1', '']
Error parsing line [76503]: ['U209006634', '491.21/17', '']
Error parsing line [76504]: ['U209024591', '692.9', 'P']
Error parsing line [77229]: ['U208003535', '787.20', 'S']
```

Once the information is read in raw format and stored in a variable it becomes necessary to arrange it. The data is processed to define the arrivals per hour and it is then, simplified into two main strips. A header is written with all the fields required and the data is introduced with the required delays (note that the iterations when creating the output file start at a value of 7, because the biggest delay is 7 days). Finally, the data set is written in a file.

```
capacityIn = flowPerHour(dataBase, date(2010,1,1), date(2014,1,1))
stripInput = computeStrip(capacityIn, 7, 14)
totalInput = computeStrip(capacityIn, 0, 24)

strip = []
outputFile = open('/Users/sergi/Dropbox/tfg/data/data_all.csv', 'w')
row = ["A1.0", "A1.1", "A1.2", "A1.3", "A1.7", "T1", "T2", "T3", "T7","T0"]
strip.append(row)
for idx in range(7, len(stripInput)):
        row = []
        row.append(stripInput[idx])
        row.append(stripInput[idx - 1])
        row.append(stripInput[idx - 2])
        row.append(stripInput[idx - 3])
        row.append(stripInput[idx - 7])

        row.append(totalInput[idx - 1])
        row.append(totalInput[idx - 2])
        row.append(totalInput[idx - 3])
        row.append(totalInput[idx - 7])

        row.append(totalInput[idx])

        strip.append(row)

writeFile(outputFile, strip)
outputFile.close()
```

4.3. Descriptive analysis

4.3.1. Preview of the system

The Hospital Universitari de Girona Dr. Josep Trueta is a health public centre managed by the Institut Català de la Salut (ICS). The centre offers assistance to the most of people from Girona. It covers all the basic medical and chirurgical specialties, characteristic of any tertiary hospital. The emergency department receives more than 70000 admissions each year and 190 in average per day.
There are five basic types of typologies the patients can present when they arrive to the emergency department:

- Type 1: triage I. Resuscitation. Situations where resuscitation is required. Vital risk.
- Type 2: triage II. Emergency. High urgent situations. Predictable vital risk.
- Type 3: triage III. Urgent. Situations of potential risk.
- Type 4: triage IV: Less urgent. Potentially complex situations without vital risk.
- Type 5: triage V: Not urgent. Situations that allows delay of attention. No risk detected.

Despite the organization by types, it's interesting to keep in mind that each category seen above can be part of another medical classification:

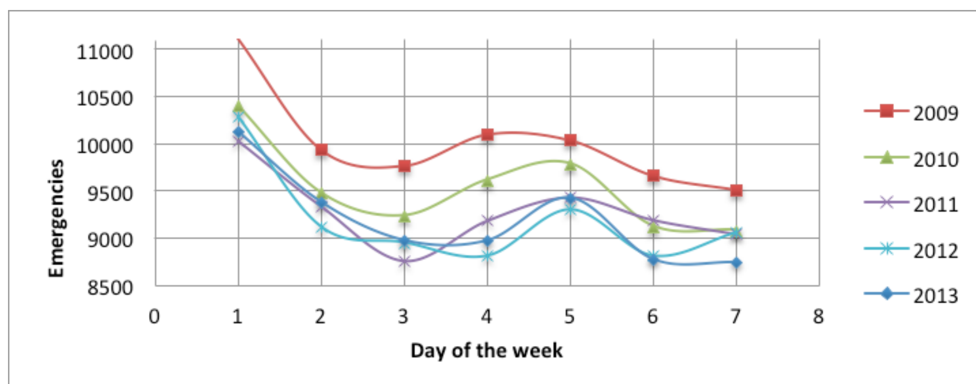
- Trauma: study of the injuries or alterations of the locomotor system.
- Pediatrics: study and treatment of infant injuries and diseases.
- Gynecology and obstetrics: study of the female reproductive system and the pregnancy, birth and puerperium.
- Medical surgical: all the rest of the medical cases.

As a more general classification:

- Respiratory: presenting injuries or alterations of the respiratory system.
- Cardiovascular: presenting injuries or alterations of the cardiovascular system.
- Organic: general injuries, everyone that does not fit neither of the groups shown above.

4.3.2. Characterization of the system

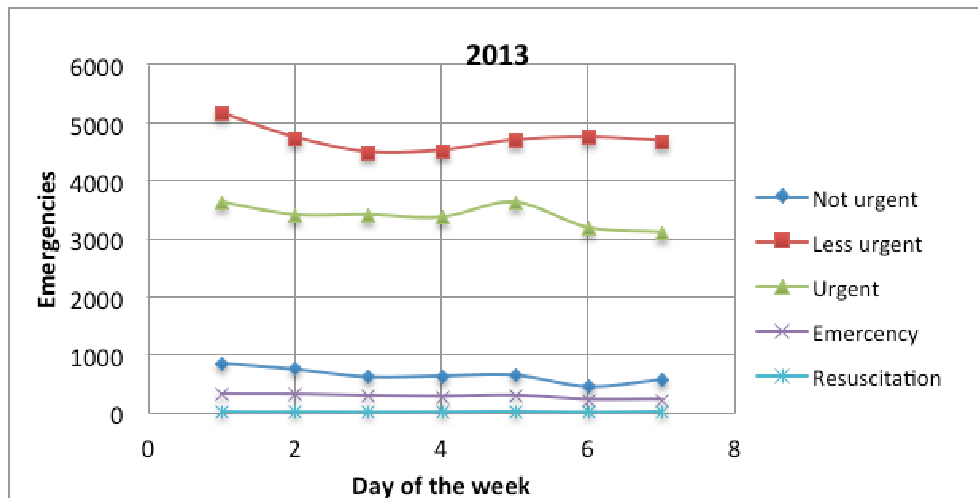
As a first approach it's interesting to visualize the flow in the emergency department along a week (from Monday to Sunday).



Plot 1. Number of emergencies per day of the week

Mondays are the most crowded days as some people wait for recover during the weekend but if they could not make it, they have to go to the hospital the first day of the week. Also along the weekend most of the primary attention centres are closed.
The second pic of demand, it is placed on Fridays (specially the last years 2012, 2013, as people do not want to lose labour days in the emergency department because of the crisis).

It is also interesting to appreciate the general distribution of the different typologies and its comparison.

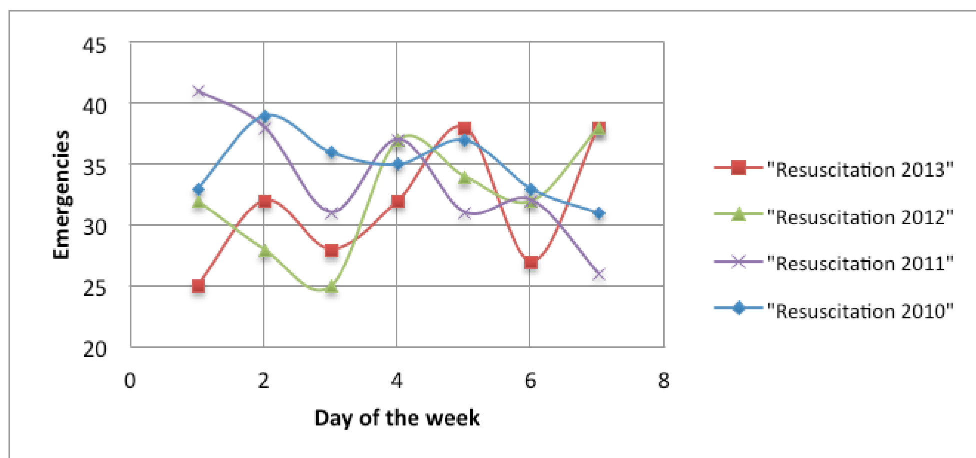


Plot 2. Number of emergencies according its typology per day of the week

The plot shows the data for the year 2013 and it has approximately the same behaviour for the rest of the years:

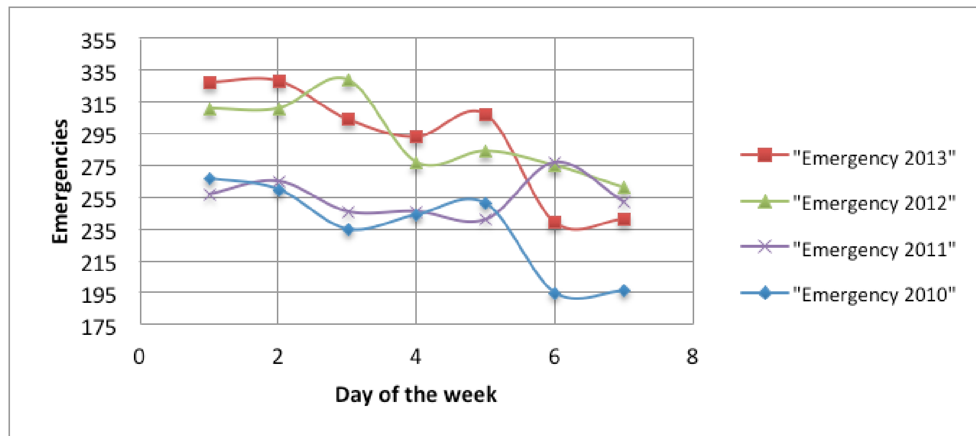
- Typologies III (Urgent) and IV (Less urgent) are the most representative ones. Having this amount of typology III shows the hospital behaves correctly (there are hospitals with high demand from IV and V).
- The amount of cases I (Resuscitation) and II (Emergency) have the lower rate and as will be seen, its distribution it's completely random.

As there are four complete years of data, could be useful to observe each typology distribution for each year studied.

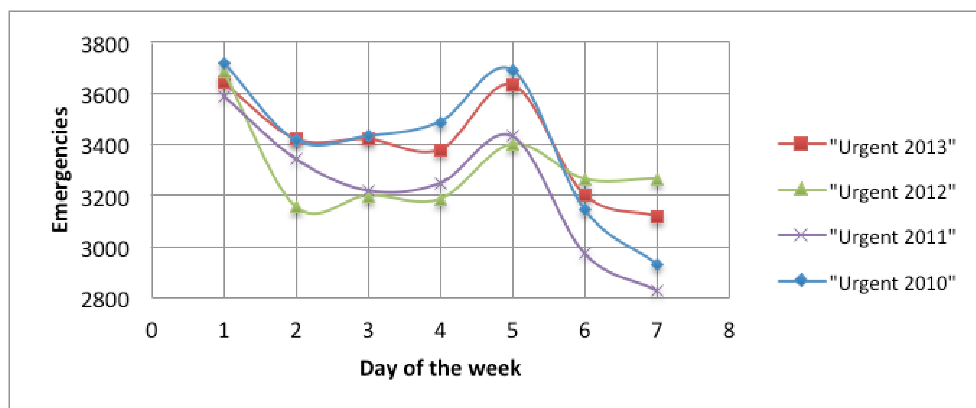


Plot 3. Number of emergencies (Resuscitation) per day of the week

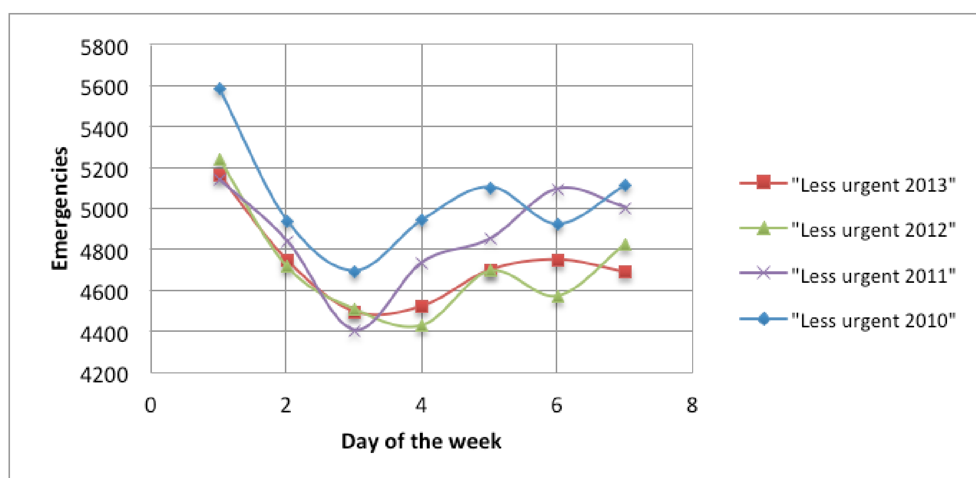
ETSEIB



Plot 4. Number of emergencies (Emergency) per day of the week

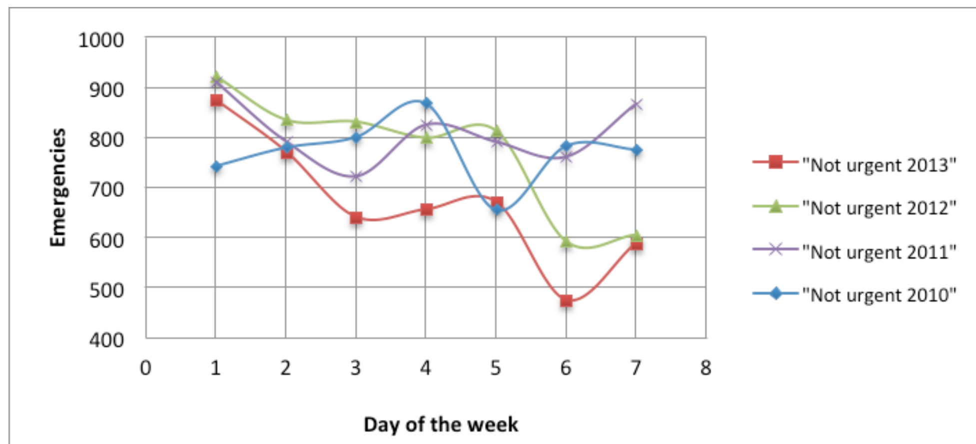


Plot 5. Number of emergencies (Urgent) per day of the week



Plot 6. Number of emergencies (Less urgent) per day of the week

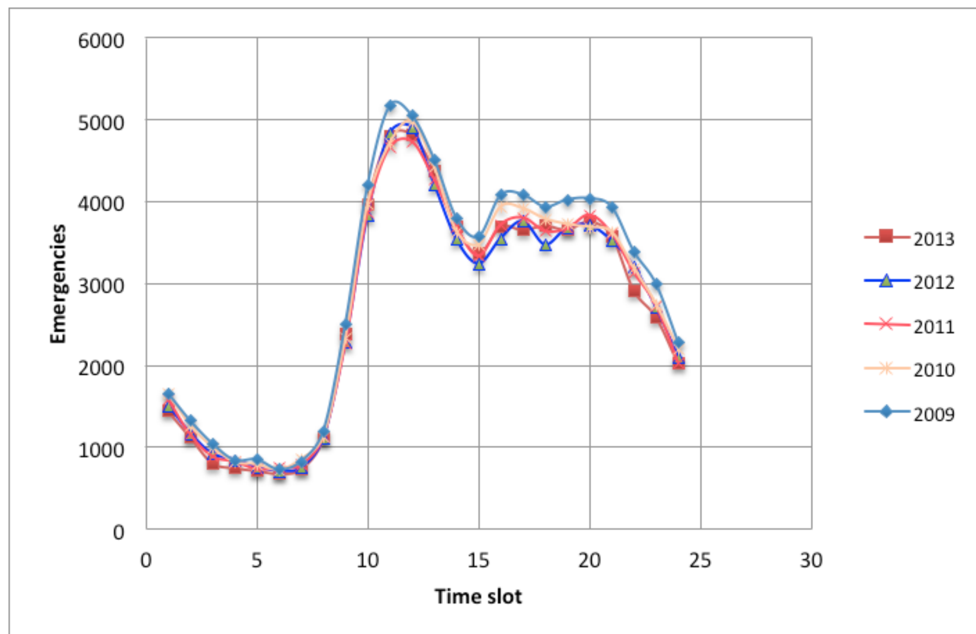
.



Plot 7. Number of emergencies (Not urgent) per day of the week

The most interesting conclusion, observing the plots above, it's how the less random typologies (especially III and IV) follow carefully the average week distribution, as the other ones, with a higher random component, seems to behave quite chaotically. This is a really important observation when it comes to build a model that suits the real data, as if only the less random typologies are taken the algorithm should get better results.

Another interesting study consists in determining how the arrivals are distributed along the day. In order to visualize this information the data is organized in time periods of one hour.



Plot 8. Number of emergencies per time slot

From the plot above is clear that despite the year studied, the general flow of arrivals, follows quite similar patterns. Is interesting to highlight how the flat curve in the early morning hours, is followed by a pronounced rise among the morning and midday and after reaching the morning peak, a slight fall followed by a quite chaotic afternoon oscillating between the same values, and finally another decrease to get to the night hours.

4.4. Time series analysis

Once the descriptive analysis was completed, as a first approach to solve the problem, a time series model was set. Even though it is not going to be used for getting suitable solutions for the current project, seems a good idea to summarize the steps that have been followed using the ARIMA methodology as well as a brief resume of the results achieved. The student Eduard Barroso has been engaged with the Hospital Universitari de Girona Dr. Josep Trueta due being working as an intern during the summer of 2014. All along this period, despite analysing in detail the volume and occupation in the emergency department, a times series analysis has been performed in order to be able to forecast the volume of emergencies. In the current chapter (2.4), only the most useful conclusions are explained.

Time series analysis embraces methods for analysing time series in order to extract meaningful statistics and other characteristics from the data. Time series forecasting is the use of a model to predict future values based on previously observed values.

The studied data, which clearly follows a stochastic pattern, it's build up by a sequence of random variables evolving through time. It must be said that the methodology used for work with time series does not require a huge amount of computations but does need repeated calculations involving a great number of variables. Because of the reasons presented, the analysis it's made using free statistical software called RStudio.

Methodology used:

- Model identification: study of the main time series components (trend, seasonality, cycles and random).
  Trend: there is a long-term increase or decrease in the data.
  Seasonal: series influenced by seasonal factors. Seasonality is always of a fixed and known period.
  Cyclic: when the time series exhibits rises and falls that are not of fixed period. The duration of this fluctuations is usually, at least, of two years.

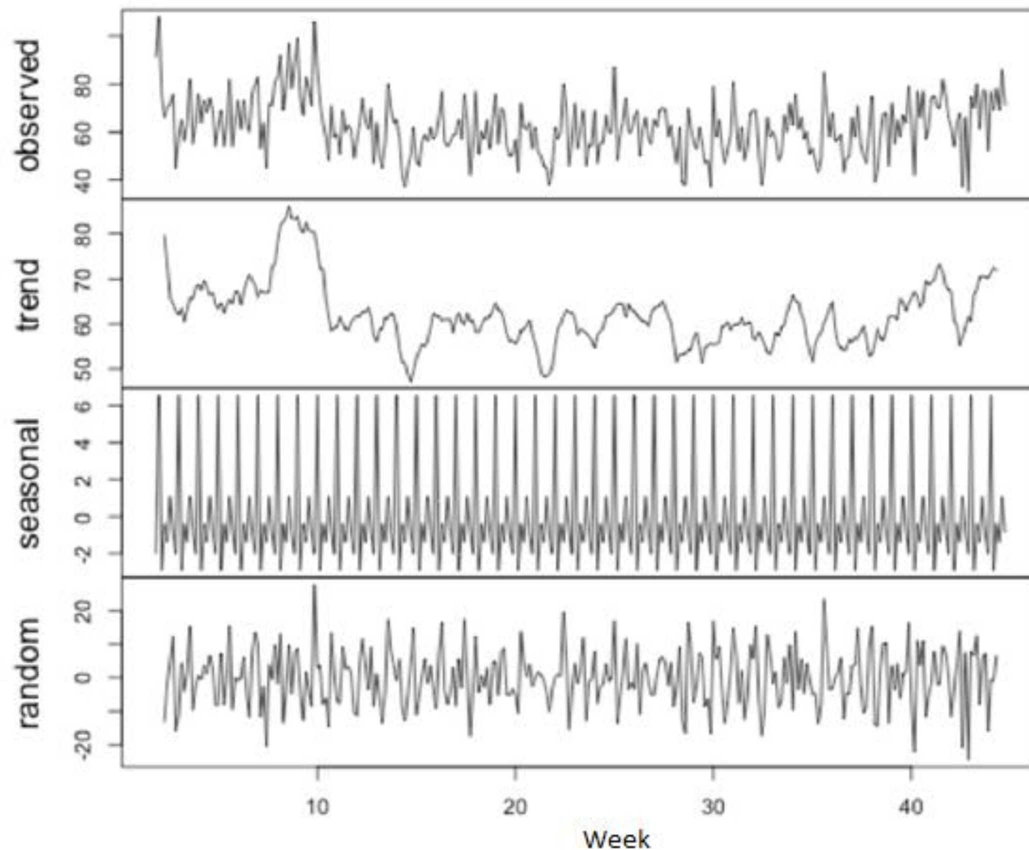
- Model fitting: using R functions and corroborating the results with the correlograms.
- Model tuning and prediction.
- Error testing.

```
1   #Data Reading
2   urg=ts(read.table("Urg2012.txt"),start=c(1,7),freq=7)
3   plot(urg,main="2012", ylab="Emergencies", xlab="week")
4   monthplot(urg,main="2012", ylab="Emergencies", xlab="day of the week")
5   #Differenciation?
6   d7urg=diff(urg,7)
7   plot(d7urg)
8   d1d7urg=diff(d7urg)
9   plot(d1d7urg)
10  var(urg)
11  var(d7urg)
12  var(d1d7urg)
13  #ARIMAmodel
14  acf(urg, lag.max=60)  # plot a correlogram
15  acf(urg, lag.max=60, plot=FALSE) # get the autocorrelation values
16  pacf(urg, lag.max=60)  # plot a partial correlogram
17  pacf(urg, lag.max=60, plot=FALSE) # get the partial autocorrelation values
18  auto.arima(urg,ic="aic") #Find "best" fitting need FORECAST package
19  urgarima<-arima(urg, order=c(1,0,1), seasonal=list(order=c(1,0,1), period=7))
```

In the first section, *Data Reading*, the data for each year can be loaded and set as a temporal series for further analysis. Then, with the help of several plots, like *monthplot* or *decompose* the features (trend, seasonality, randomness) of the series can be analysed. In case it presents a clear trend, it can be erased from the model by applying the differentiation (second section). As this whole sections represents a sneak peak of the time series analysis, the ARIMA model, has been fitted using the *FORECAST* R package.

In the nineteenth line of the last section, the first ARIMA model fitted, has been chosen according to the results of eighteenth line (*auto.arima*) and its correlograms.
Another interesting function it is the one that decomposes the time series in the different intrinsic components (*decompose*).

```
20  plot(decompose(urg))
```



Plot 9. Time series components for the year 2012 (Decompose function applied)

- Parameters estimation: fitting the data with the best ARIMA model according to AIC (Akaike information criterion) as the parameter to be optimized.

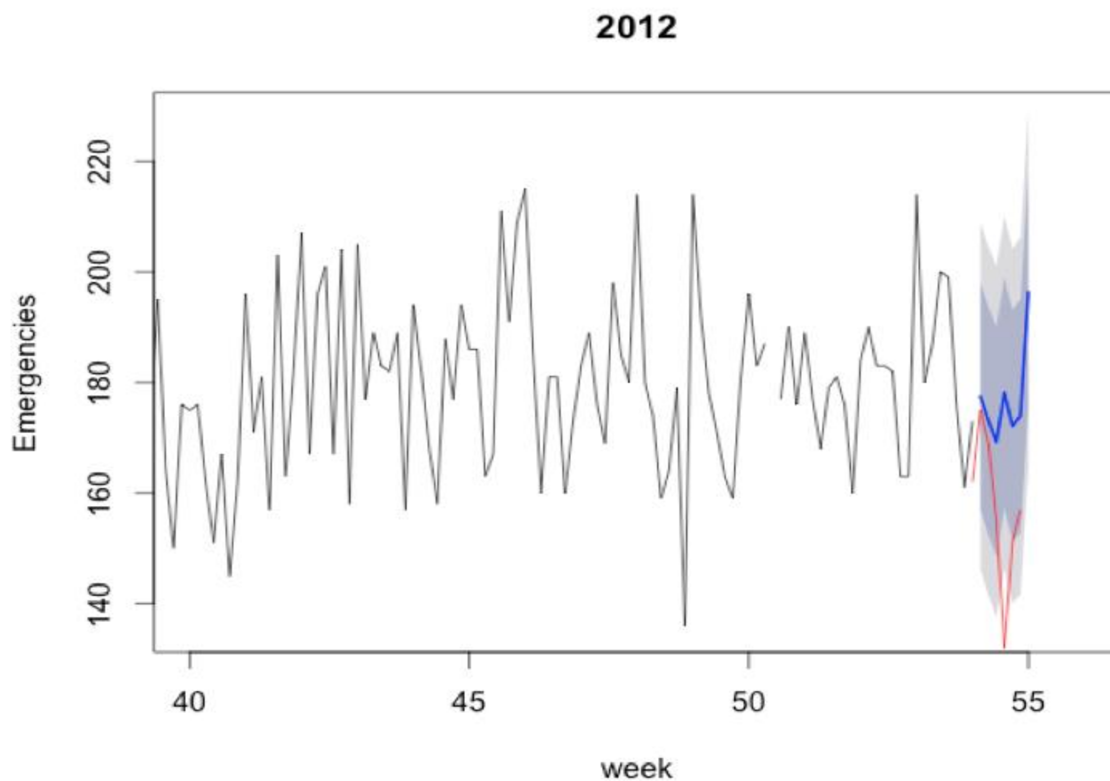
```
21  ### Get rid off atypical values(in order to improve the model)
22  ##Days arranged from biggest to lowest residuals
23  resid=resid(urgarima)
24  idx=order(abs(resid),decreasing=T)
25  resid[idx[1:10]]
26  library(chron)
27  (dates("1/1/2012")+0:364)[idx[1:10]]
28  urg.lin=urg
29  urg.lin[idx[1:5]]=NA
30  #New Variables
31  Tmax<-read.table("Tmax2012.txt", header=TRUE)
32  Tmin<-read.table("Tmin2012.txt",header=TRUE)
33  vacation=rep(0,length(urg))
34  vacation[213:243]=1
```

Combining the variables in different ARIMA models allows us to decide which the better ones are in order to fit the model.

```
35 #New model
36 urgarimanew<-arima(urg.lin, order=c(1,0,1), seasonal=list(order=c(1,0,1), period=7),xreg=data.frame(Tmax,Tmin,vacation))
37 urgarimanew<-arima(urg.lin, order=c(1,0,1), seasonal=list(order=c(1,0,1), period=7),xreg=data.frame(Tmax,vacation))
38 urgarimanew<-arima(urg.lin, order=c(1,0,1), seasonal=list(order=c(1,0,1), period=7),xreg=data.frame(Tdiff))
```

After analyse all the ARIMA models according to the AIC criterion, the best one is chosen to fit the real data. Next step consists in make the predictions with the selected model. Also it is interesting to plot the results of the prediction together with the real data.

```
40 #Forecast
41 library(forecast) # load the "forecast" R library
42 urgforecast<-forecast.Arima(urgarimanew,h=7,xreg=data.frame(Tmax=rep(15,7)))
43 plot.forecast(urgforecast,xlim=c(40,56))
44 ###Add observations
45 obs1yearlat=ts(read.table("Urg2013.txt"),start=c(1,1),freq=7)
46 obs2=obs1yearlat[1:7]
47 obs=ts(obs2,start=c(54,1),freq=7)
48 lines(obs,col=2)
49 pred=urgforecast$mean
```

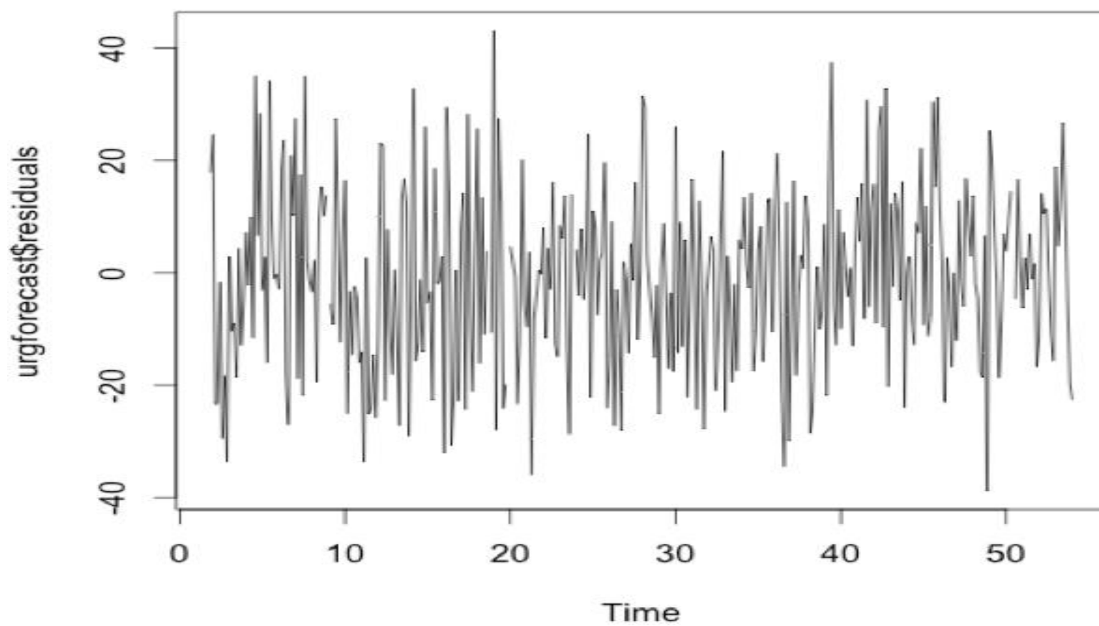


Plot 10. Predictions (blue line) and real values (red line)

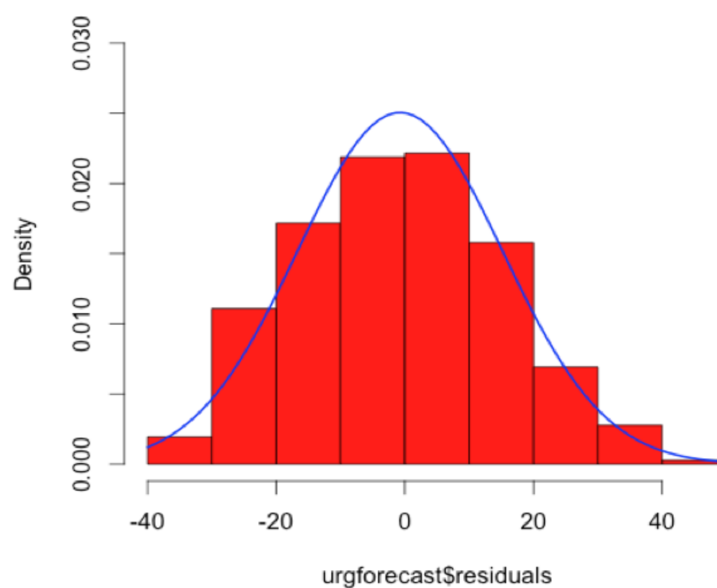
The prediction of the first week of 2013, as observed, it is quite disappointing as it has most of the values outside the 85% interval confidence, and even some of them outside the 95%.

- Hypothesis validation (Ljung-Box methodology): study of the residuals.

```
50 ##RMSER
51 RMSER=sqrt(sum(((obs-pred)/pred)^2)/(length(pred)-1))
52 #Comprovation
53 acf(urgforecast$residuals, lag.max=60,na.action=na.pass)
54 Box.test(urgforecast$residuals, lag=60, type="Ljung-Box")
55 plot.ts(urgforecast$residuals) # make time plot of forecast errors
56 plotForecastErrors(urgforecast$residuals) # make a histogram
57 hist(urgforecast$residuals,prob=T,col=2,ylim=c(0,0.03))
58 m=mean(urgforecast$residuals,na.rm=T)
59 s=sd(urgforecast$residuals,na.rm=T)
60 curve(dnorm(x,mean=m,sd=s),col=4,lwd=2,add=T)
```



Plot 11. Residuals of the time series analysis



Plot 12. Density of the residuals

Observing the Box test and the residuals distribution seems that the model is correctly adjusted. However, the predictions obtained are not good for a forecast exercise.

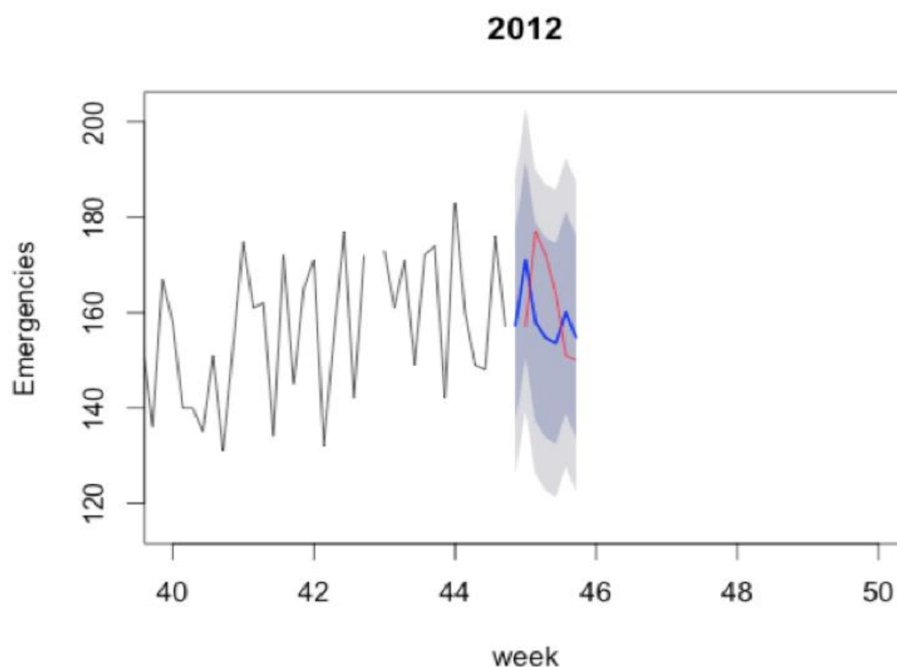
4.4.1. Conclusions for the Time Series analysis

After several tests using ARIMA models, seems that the data studied it is not susceptible to be modelled by this method. Its erratic component does not let the forecast perform close to the reality. In order to try to lower the random phenomenon in the process, different assumptions are taken:

- Model only the typology of patients that do not follow a random distribution (like types I, II and V), such as types III, IV.

- Predict spots amongst the year, and not use the whole year in order to forecast the first week of the next year as this is located in a critic dates.

With the dictated assumptions, new results are found. The best one is the follow one:



Plot 13. Predictions (blue line) and real values (red line)

In this case, using only the less random types of medical typologies (III and IV), and forecasting between the year (in this case a week among October), the results are notably better.

Anyway, this is the best case-scenario found among many tests and models used. Also with the assumptions taken, especially the one to take a noncritical week doesn't offer a good performance of the generalized model.

4.5. Artificial Neural Networks

The artificial neural networks (ANN) are learning algorithms, representing a large branch of the machine learning, which are inspired by the structure of the human brain. This represents the leading non-linear statistical data mining tool currently used. They are useful to model complex relationships between inputs and outputs.

In this project the three R most used packages are going to be chronologically evaluated and tested in order to explore each one capabilities towards solving, in the best way, the current project. After a critic discussion, the R Stuttgart Neural Network Simulator (RSNNS) package has been chosen for develop the final solution, therefor all the mathematic descriptions of the algorithm are kept for the reference 2.5.3 where this package is explained in detail.

4.5.1. Neural Net package

The first artificial neural network developed, using the Neural Net R package, considers that only 4 variables are relevant to determine the number of inputs in the second strip of a given day. Those variables are: the number of admissions in the first strip of the day, the day of the week, the flu incidence and the holidays; some of those, are qualitative variables that have to be rewritten as quantitative variables. For this reason, 9 dummy variables are needed: 7 for the day of the week, 2 for the flu incidence and 2 for the holidays. It is important to mention that the variable *holidays* considers also one day before and after the holiday date.

One of the biggest issues when working with *dummy* variables is the significant increase of inputs to the system, which has a computational impact on the precision. It is important to remember that the number of inputs can be increased if the train data set is big enough. In this case the data set has 1096 rows –3 years of data.

4.5.1.1. ANN first font code

The input data is read from a *csv* file containing the following fields: admissions in the 1st strip, day of the week, flu incidence, holidays and admissions in the 2nd strip.

```
# Read data in .csv format.
# Headers: entries_1 (#), day_of_week, flu (0-1), holiday (0-1), entries_2
(#)
raw_data = read.csv2(file="v1_10_type_3_4.csv", header = F,sep = ";")

# Load library nnet
library(nnet)
set.seed(1)

# Set inputs, atribut from 1 to 4
inputs <- as.matrix(raw_data)[,1:10]
```

```
# Set outputs, atribut 5
outputs <- as.matrix(raw_data)[,11]
```

A data set is defined as well as a data train and a test set. The test set is extracted randomly from the data set with a length of 100 rows.

```
# Create dataTrain, dataSet
dataSet <- data.frame(cbind(inputs, outputs))
idx = sample(1:nrow(dataSet), 100, replace = F)
dataTrain <- dataSet[-idx,]
testSet <- dataSet[idx,]
```

The names of the different fields are read and the formula to introduce to the ANN is defined according the name of the parameters.

```
# Define names of the variables
n_in <-names(train)[1:10]
n_out <-names(train)[11]

# Define formula
f  <-  as.formula(paste(paste(n_out,"~",  collpase=NULL),  paste(n_in,
collapse = " + ")))
```

The ANN is trained with 10 computational units, a linear output and a range equals to 0.9.

```
# Train nnet
nn <- nnet(f, size=10, data=dataTrain, linout=T, rang=0.9)
plot(nn$fitted.values)
```

To check the accuracy of the ANN model, the predictions are compared to the real values. The error is computed.

```
# Validate with the test set
predictions <- predict(nn,testSet)
observations <- testSet$outputs

# Compute error
rmsd<-0
for (i in 1:length(idx)){
 rmsd<-rmsd+(observations[i]-predictions[i])^2
}
RMSD<-sqrt((rmsd)/length(idx))
results<-cbind(predictions,observations)
```

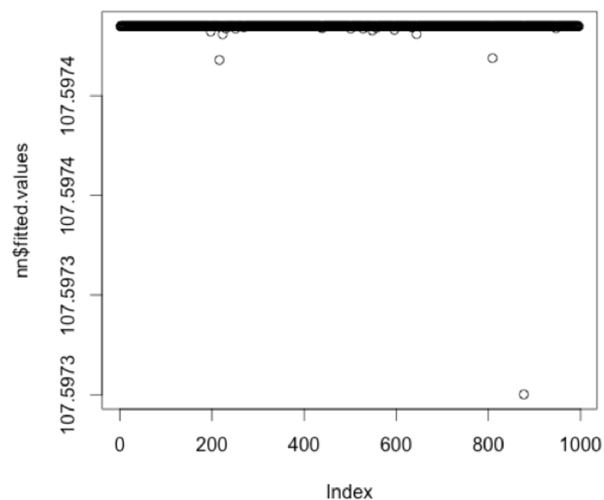
4.5.1.2. ANN first predictions

Once the neural network it's computed, it's interesting to see how many weights and iterations were required for the algorithm to converge and offer a concrete solution.

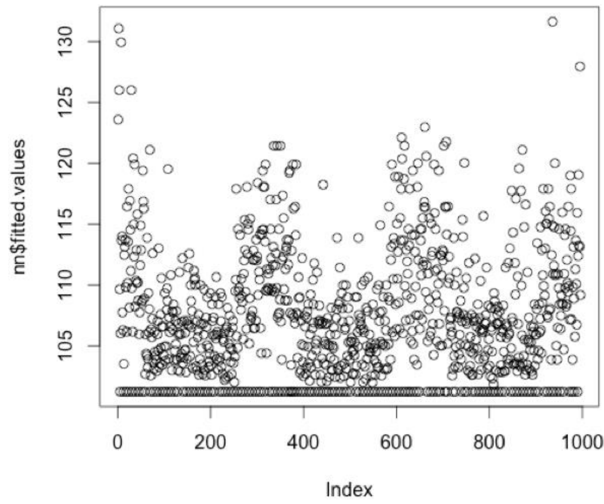
```
> nn <- nnet(f, size=10, data=dataTrain, linout=T, rang=0.9)
# weights:  121
initial  value 11250330.058462
iter  10 value 175801.664693
iter  20 value 173363.396914
iter  30 value 172205.241401
iter  40 value 166562.793733
iter  50 value 154228.285525
iter  60 value 149999.511491
iter  70 value 149185.318145
iter  80 value 148662.924314
iter  90 value 148478.313792
iter 100 value 148468.132260
final  value 148468.132260
stopped after 100 iterations
```

In this first analysis, it is useful to plot the *nn$fitted.values* in order to appreciate if the values fitted by the neural network are correct or otherwise the algorithm it's just making predictions of the mean.

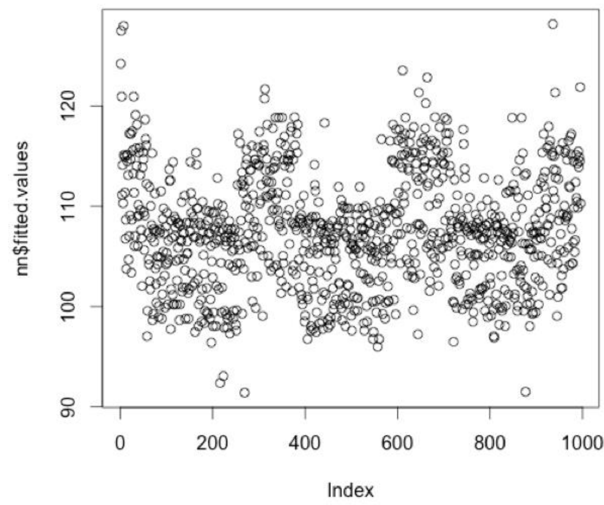
```
> plot(nn$fitted.values)
```



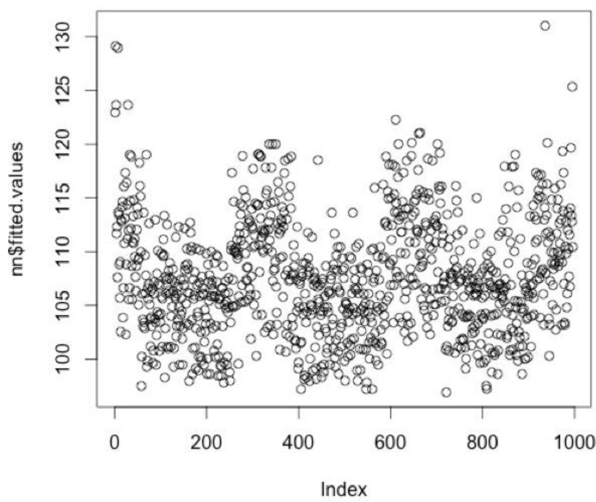
Plot 14. First training example



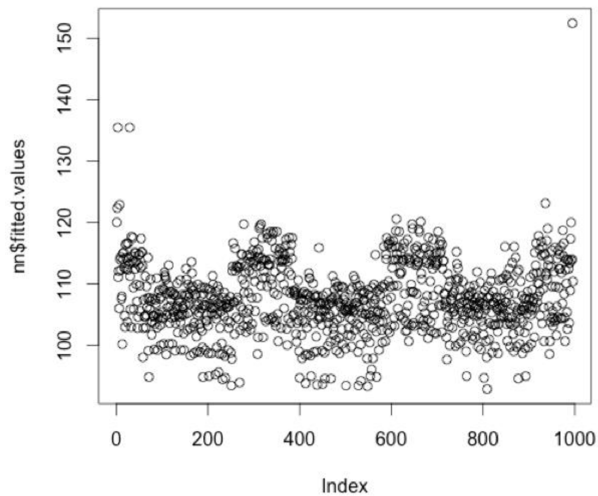
Plot 15. Second training example



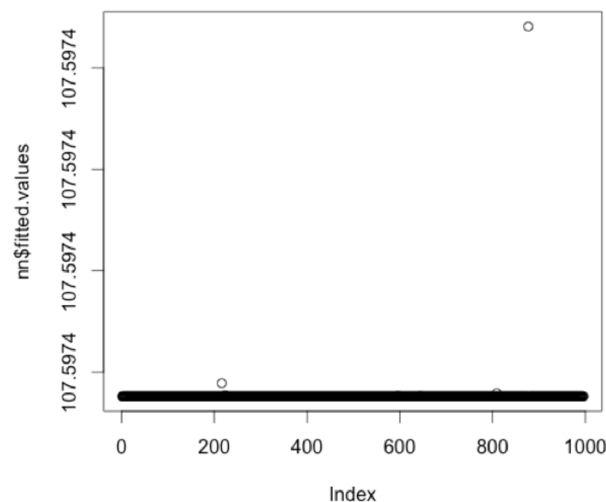
Plot 16. Third training example



Plot 17. Fourth training example



Plot 18. Fifth training example



Plot 19. Sixth training example

After each iteration the fitted values change according the new weight values obtained with the algorithm. The first computation is working clearly wrong as it predicts for every trained value, the same output, i.e. same prediction. Once the iterations are processed, the algorithm works each time better since it reaches the best-fit possible and starts to mask the weights until predicts, again, the same fix value (around the mean value).

Root mean squared deviation it is used in order to understand, numerically which of the cases is fitting the values with the accuracy required. Using this methodology, errors from more than 50% (for the first and last iterations) and less than 15% (best approaches with a relative error around the 9,5%) are found.

4.5.1.3. Nnet package

As a first approach to work with artificial neural networks with real data, the *neuralnet* package from R packages was chosen. It offers a huge amount of possibilities when it comes to select the best parameters for the network, also it allows to work with more than one hidden layer with the required processing units among other options. After several number of trials, and obtaining bad results as the ones shown in the plots above (it makes predictions based on the mean of the outputs, most of the times), it leads to change the library to work with.
As a reference from Prof. Marc Saez, PhD, Cstat CSci Research Group on Statistics, Econometrics and Health (GRECS), and due obtaining good results thanks working with the R library *nnet*, it is decided to change the library.
The *nnet* consists in a package which offers less number of parameters and options. This means that there are several information that is fixed by default. According with the references, by using the new library allows to perform better predictions than the previous one. It is change, as the main reason, because using the *neuralnet* package there exists a strong responsibility to set the large amount of parameters and combinations whereas in the other option this responsibility decreases significantly. So, as a first approach working with artificial neural networks it make sense to start choosing less parameters and options.
Despite obtaining better results, it does not offer enough accuracy and the results do not offer any consistency or reliability.
Finally it is thought that, in part, the problem could be related with the data used as the input for the training of the algorithm. There are several references related to work with these artificial intelligence methodologies that indicates that in some cases, the standardization of those input variables is fundamental as a key to improve the performance. Also it would be really interesting to be able to set all the parameters and monitor them throughout all the tests. Working with the software RStudio, hence all the functions are inside libraries and sometimes the environment it is quite closed, it is decided to start coding the same algorithm but in python. By having the neural network coded in python, there is the chance to analyse and choose the best possible values for all the parameters in case it is required. Also it has to be in Python in order to set the executable application mentioned in the scope of the project.

4.5.2. Nnet package

After the results and conclusions from the first approach, it is decided to use the *nnet* package, setting the following parameters:

- **size**: number of units in the hidden layer. Can be zero if there are skip-layer units. It allows to mimic, in order to fit, the non-linear processes that are being analysed. The tests have been performed using a number of processing units between the number of inputs and 100, obtaining the best results for 20-25.
- **data**: Data frame from which variables specified in formula are preferentially to be taken. The corresponding Data train set has been used.

- **linout**: switch for linear output units. Default logistic output units. The predictions are not meant to pursue a classification, so there is the need of using a linear output.
- **maxit**: maximum number of iterations. Default 100. When processing the algorithm, if there is no convergence, the analysis stops when reaching the 1000 iterations in the current algorithm.
- **decay**: parameter for weight decay. Default 0. Over-fitting is a recurrent problem when using neural networks, it is useful to set the decay different to 0, as it penalizes large weights, so partially preventing from over-fitting. In our case is set to 1e-4.
- f**ormula**: a formula of the form class ~ x1 + x2 + ...

Another important change that has been made consists in standardized all the variables before they are being applied into the artificial neural network algorithm. The standardization can be calculated with the following code:

```
train<-matrix()
first = 1
for(q in (1:ncol(train_old))){
  m<-mean(as.matrix(train_old[q]))
  s<-sd(as.matrix(train_old[q]))
  norm<-(train_old[q]-m)/s
  norm
  if (first == 1){
    train<-norm
    first = 0
  }
  else{
    train<-cbind(train,norm)
  }
}
```

Once the basic structure of the neural network is set, is time to test it by using different input variables. Pursuing the goal of work with the best possible input variables, several references (books, papers, etc.) had been kept in mind. A remarkable fact, is to be able to use all the knowledge and expertise from people having high ranks inside this concrete field. According both sources, the most used variables for modelling the volume of arrivals at the emergency department, are the following ones: Temperature, Humidity, Vacations/Holidays and if there are some kind of big social event like an important football match for example.

After obtaining all the variables required and applying them, the artificial neural network results are notably improved as it allows to predict with less than 14% of RMSE (relative mean square error). Therefore, there is still a problem as it is not possible to predict with less than 9.5% error in any case. This is the main reason to try to solve the problem applying a concrete methodology from scratch.

4.5.2.1. Disaggregate the information

When it comes to deal with a process that is the result of the aggregation of different non-linear features and its interactions, it may be useful to break up those inputs or parts of the process in order to try to lower its complexity.

During this project the following assumptions have been made for, by understanding the sub-processes, improve the accuracy of the predictions:

- Instead of using the *n* past days to forecast the following day, the information of the current day is used. By partitioning the day in two different strips, information of the first strip of the day, can be used to forecast the last strip of the same day. And also allows, like the previous case, the use of delayed information.

- Once it is verified that using the total number of arrivals at the emergency department to make the predictions does not offer the desired results, the new approach consists in disaggregate the total amount of patients in three basic typologies: respiratory, circulatory and organics. By analysing each of the three groups, the forecasts are notably better as the algorithm can be particularly modified and adjusted to each different situation.

- Another interesting break up could be the one consisting in disaggregate each one of the above three cases by age. Other three categories including the patients aged between 0-16 years, 16-65 years and more than 65 years. As there is not a big volume of arrivals for each type and age combination this option has been discarded in the scope of the present project.

- There is another option to disaggregate the data in a functional way. It consists in having in mind the fact that for each of the three categories shown above (respiratory, organics and cardiovascular) they can have 5 different types of patients according to its triage (emergency indicator). It has already been used for the temporal series analysis with quite good results. Even so, in this project this option has not been taken for the same reason explained above: low amount of data for each disaggregated case. Anyway it is a demonstrable fact that selecting the less random types (excluding the most urgent, and less probable type) in order to lower the randomness of the time series.

4.5.2.2. Data Sets

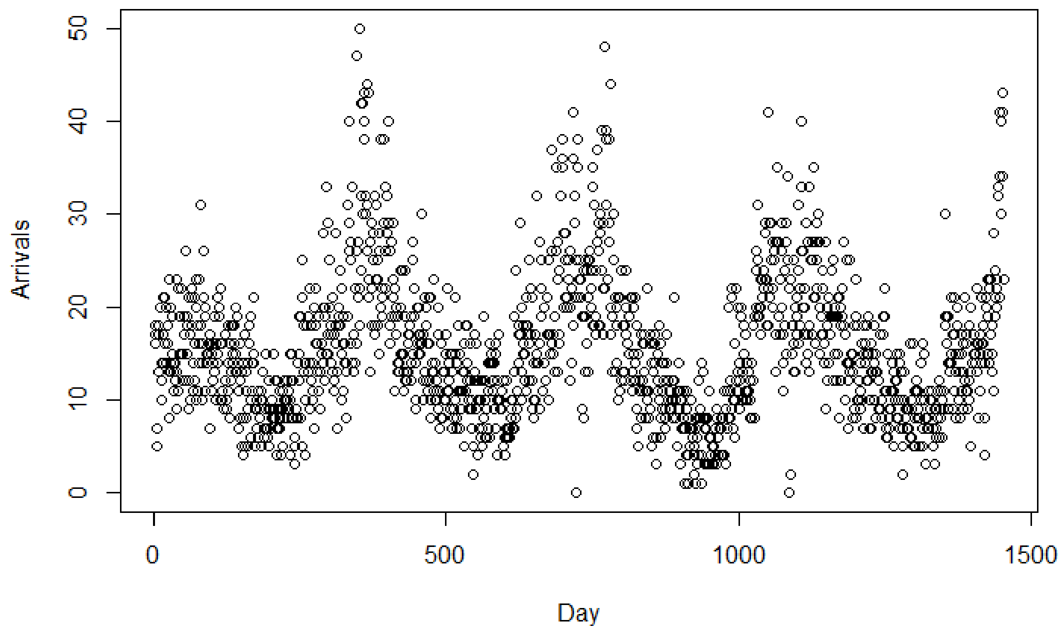
The data set used to carry out all the analysis done in the project is based in the data recorded in the emergency department of the hospital during the years 2010, 2011, 2012 and 2013. Knowing that 2012 was a leap year, we have 1461 days to analyse. Actually the study has been done with a few less days since using delayed variables, a week has been lost following that purpose (there is no possibility to have data from 2009).

Focusing on the artificial neural network training, the last 180 days of 2013 have been removed and keep as a *validation set*. The rest of data have been used for the cross validation algorithm (k=9) in order to have different sets: *data train* and *data test*. Once the best neural network

weights have been chosen, they are applied to the *validation set* data in order to get the predictions in a real trial. Afterwards it is possible to analyse the fitting power and performance of the current algorithm.

4.5.2.3. Respiratory type

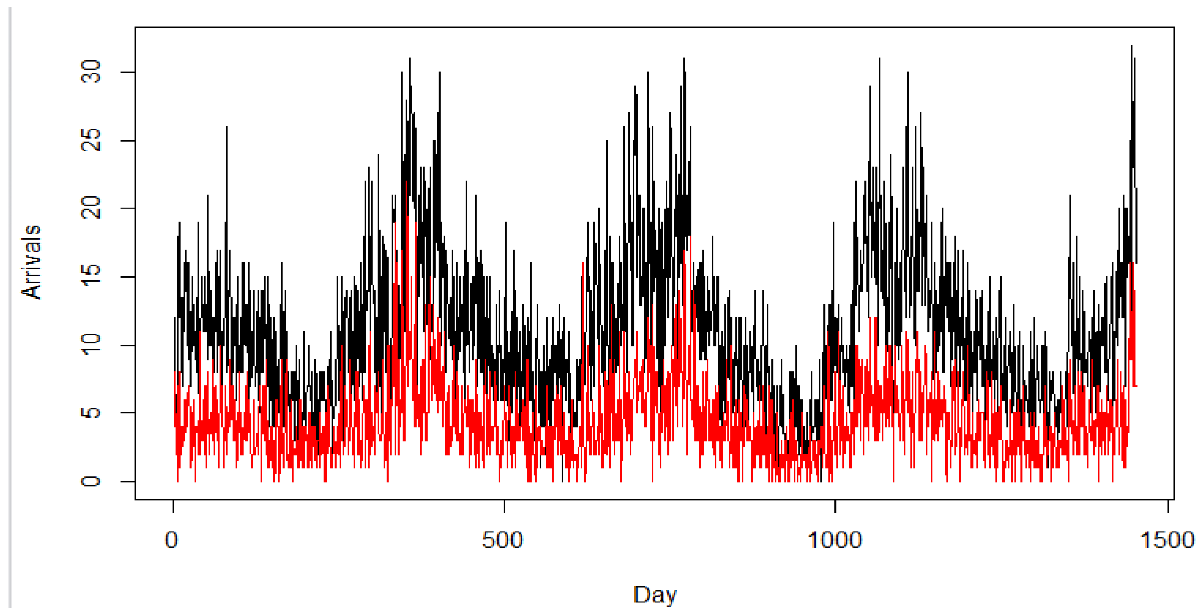
As a first approach to model the process, a plot of the total arrivals for this medical speciality vs period studied is done.



Plot 21. Respiratory emergencies per day (2010-2013)

According to the graph above, this type of patients follows a strong seasonality all along the four studied years. It helps us to notice two different phases in each year: two periods of time where the volume of arrivals notability increase (corresponding to the months of winter and autumn) and the other two corresponding to the spring and summer.

It is also interesting to plot the arrivals corresponding to the first strip during the whole studied period in order to verify if, for this particular case, it follows the same pattern (the arrivals at the first strip are coloured in red, and the ones at the second one, in black).



Plot 22. Respiratory emergencies per day (2010-2013)

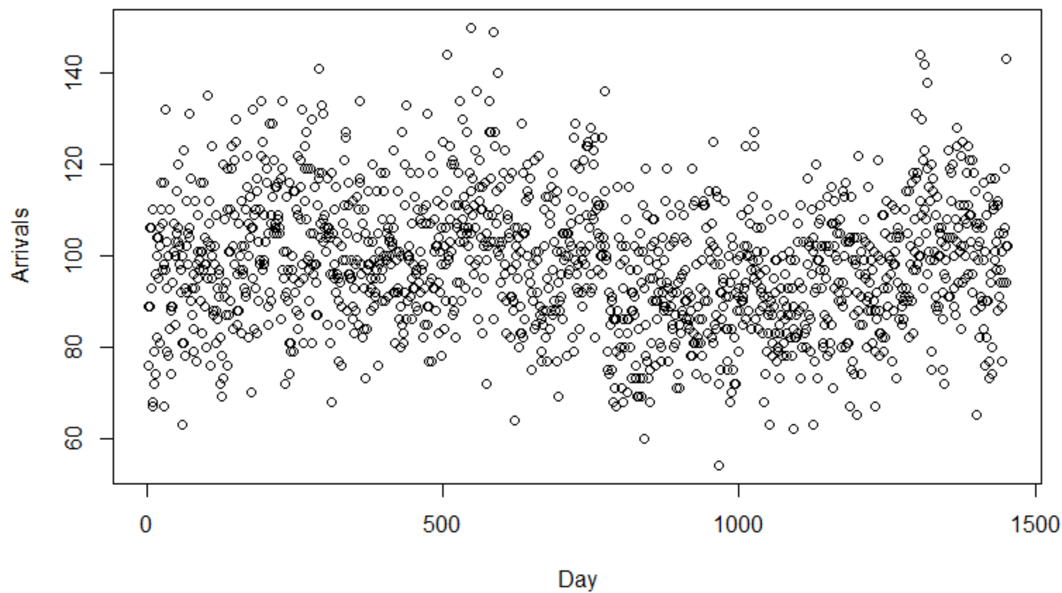
It seems that both time series follow a similar pattern. To remark this fact is useful to apply the correlation function to see if there is a real relation.

```
cor(data$A_1,data$Total-data$A_1)

[1] 0.5269408
```

This is an interesting result because due to the partition done, there are two important time series that are quite correlated, and will be an angular point to develop the algorithm towards good fitting.

4.5.2.4. Organic type

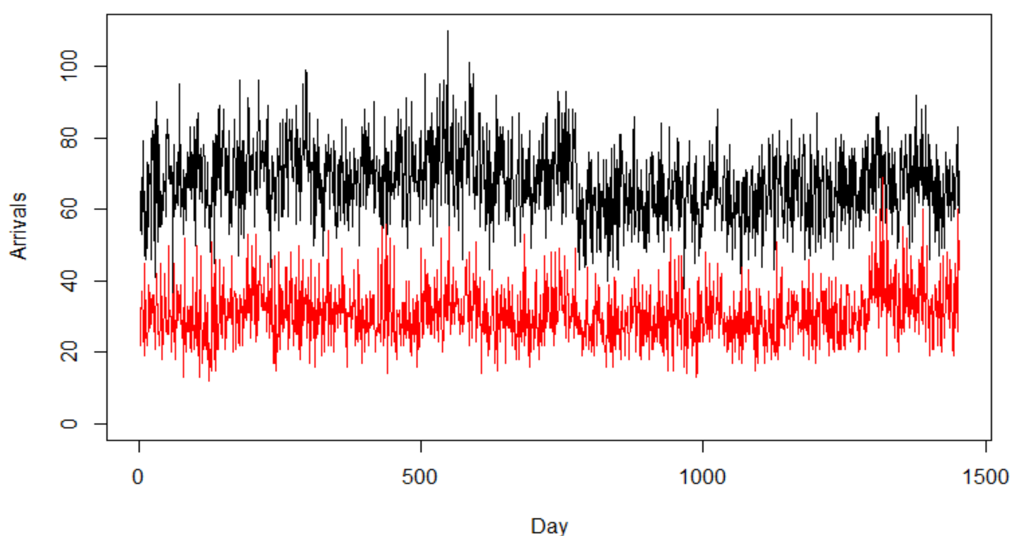
As a first approach to model the process, a plot of the total arrivals for this medical speciality vs period studied is done.



Plot 23. Organic emergencies per day (2010-2013)

In this particular case there is no seasonal or even a pattern been followed by the data. Even thought this fact looks worst in therms of the prediction performance, for this type there is a significant larger amount of events which is a good in terms of training the algorithm.

It is interesting to check the same steps as in the first case in order to make up and identify differencies between both processes.



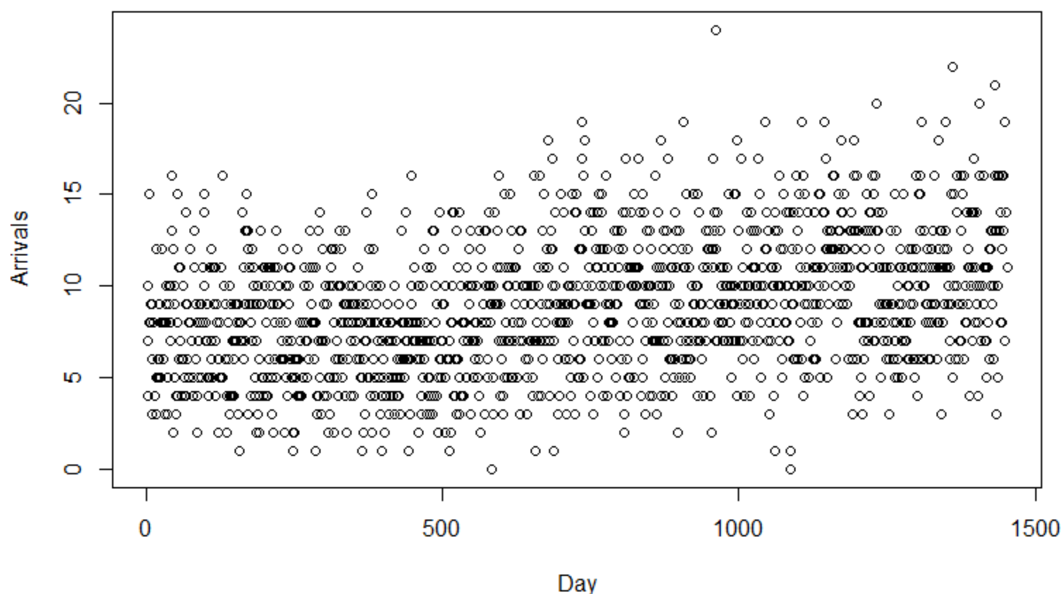
Plot 24. Organic emergencies per day (2010-2013)

In this case there is no interesting relations between both features. This conclusion is logic as long as we understand that this is the biggest group and it includes all the unclassified cases. Also for this case the correlation is much below than 0.5 (0.23).

Even though it does not show any clear relation unlike the case analysed above, it is interesting to try to model this process as it represents a complete different problem, and there is enough amount of data to try to do it.

4.5.2.5. Cardiovascular type

This last case of study shows a similar behaviour as the one seen above. The problem, in this case, is that there are not enough amount of data to try to model a good artificial neural network. It is quite disappointing as normally, for this type of patients, there is a clear seasonality as the respiratory ones.



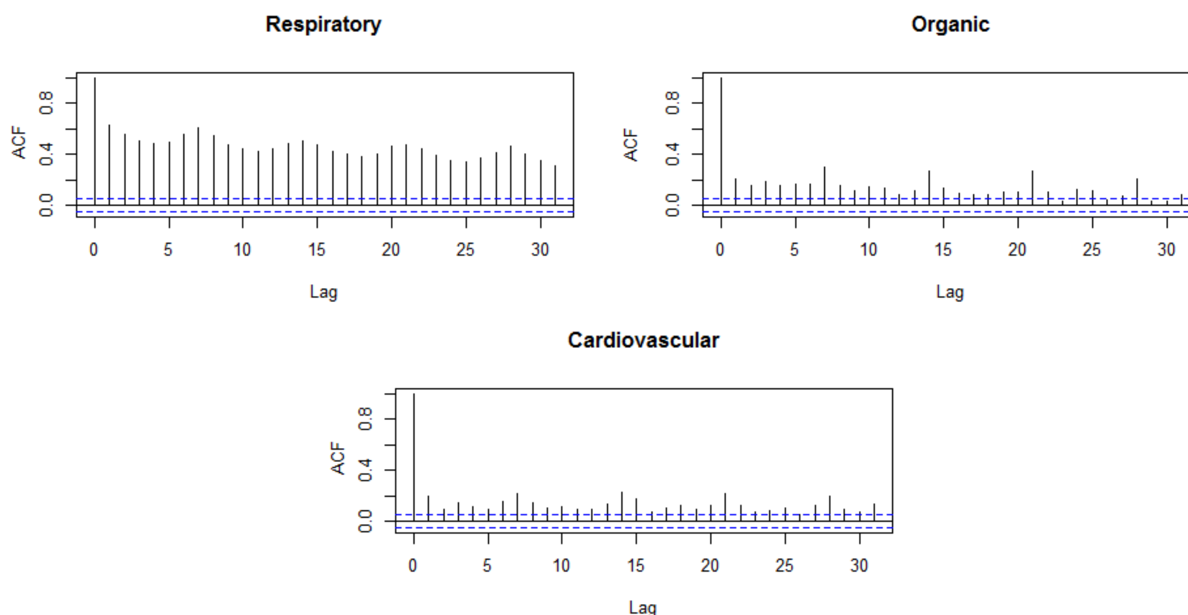
Plot 25. Cardiovascular emergencies per day (2010-2013)

Clearly, there is not enough registers to work with, and not being able to spot any clear relation, as the respiratory process, it is decided not to analyse this kind of medic speciality. Since they represent, in general, less than a 10% of the total arrivals at the emergency department it does not really altered the goal of try to help with the saturation problem.

4.5.2.6. Features for each data set

In order to know which features are better for modelling the respiratory cases (presenting strong seasonality) or for the organic ones, the same kind of features are going to be used for both processes as an initial approximation.

In order to perform this analysis, the already applied Box-Jenkins methodology is used for discover if there exists an autocorrelation or partial autocorrelation within the variables.



Plot 26. Auto covariance or autocorrelation function for the three typologies

By comparing the three plots, the relation between the seasonality of the respiratory type and the high autocorrelation presented in the ACF plot, is highlighted.

In order to notice the effect of the recent conclusions in a real artificial neural network performance, three similar data sets are set up. Each one has as input, the following covariates:

A_1	A_1 (lag 1)	A_1 (lag 2)	A_1 (lag 3)	A_1 (lag 7)	T_1 (lag 1)	T_2 (lag 2)	T_3 (lag 3)	T_7 (lag 7)	Holiday (3 days)

Table 1. Explicative variables used in the model

Where *A_1* stands for the arrivals at the first strip of the day (from 07:00 to 13:00) and *T* for the total volume of arrivals (A_1+A_2). According to papers with similar goals as the ones pursued in the present one, it is always useful to set the vacation days. In this case when there is an official vacation, it is highlighted with a one (otherwise, zero) the previous, the current, and the following day.

Using the same algorithm and parameters for the neural network, it is computed 100 times obtaining one hundred different RMSE values and applying the mean to this vector, there is the chance to notice significant differences for each of the three types (respiratory, organics and cardiovascular).

From a 100 samples for each measure, the following means are obtained:

Respiratory	2.802%,	2.810%,	2.805%
Organic	10.437%	10.246%	10.335%
Cardiovascular	25.455%	25.455%	25.565%

Table 2. Relative error for each typology

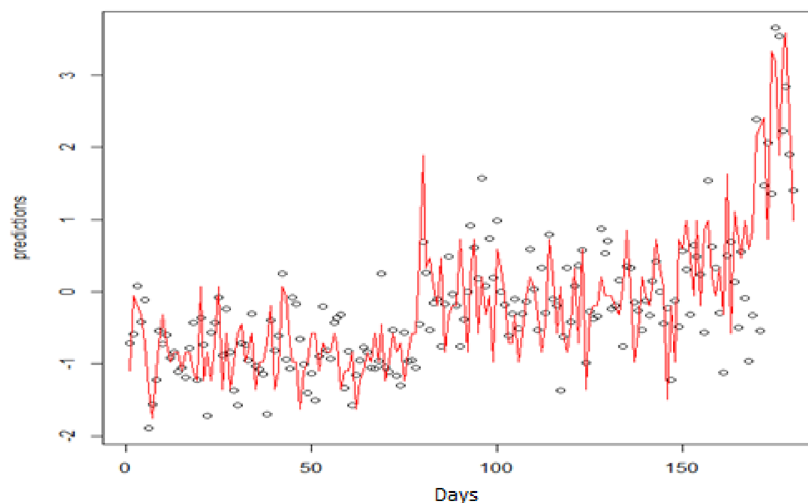
It is clear that for the type with more seasonality (respiratory category), the dynamics of the process can be highly adjusted by using the correct number of lags. The other types are performing much worst, especially in the cardiovascular case that really shows a bad RMSE mean due there are just a few cases during each weak.

To conclude, it seems reasonable to keep using the defined inputs (always standardized) for the respiratory type, the same for the organics, knowing that in order to reduce the RMSE some other variables should be introduced, and finally for the cardiovascular type it really does not look like it can be modelled by using this methodology and covariates.

4.5.2.7. Features in detail

4.5.2.7.1 Respiratory type

In this case, only using the temporal series, seems to offer quite good adjusted results for the RMSE. It is interesting to observe the plot below where the points are the predicted values for the last half of the year, and the lines in red, the real values for those days.



Plot 27. Predicted values in red and real observation in black

The RMSE value is really low, it fluctuates between 2.8% and 5%. It could be a good final result/conclusion but since there are several papers proving the existing relations between the

volume of arrivals for the respiratory type with an approximately ten-fourteen delayed days descend of the temperature, it is interesting to add this feature in order to try to improve the results.

The cold temperature index has been obtained using the following formula:

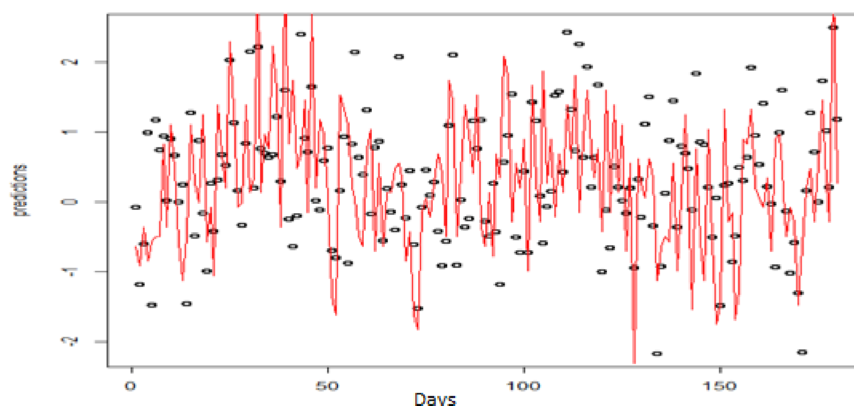
After introducing three new features, an index for the measure of the cold temperature and two delays (7 and 10 lags) of this variable, the results after several samples, are not that good as the previous ones and also, analysing the resultant graphs, it seems there is some over-fitting.

It is important to notice that since this current temporal series has a seasonal component and is offering good results, it would perform well approximations by using the ARIMA methodology.

4.5.2.7.2 Organic type

For this type, using the same kind of covariates as the previous case, the RMSE values oscillates close to the 10.5% of RMSE, so it seems a good data set to try to introduce new explanatory variables in order to reach better results. In this case, it is interesting to try it with the same cold/hot temperature index as the case above.

After trying only with the cold temperature index, the results are notability improved (8.8% error) and using the hot temperature index (its used is highly recommended in several papers) it has a better performance (8.4%), even though it is not much different.



Plot 28. Predicted values in red and real observation in black

It really does not offer such a good perspective as the respiratory type, but even though the process seems to follow a random or chaotic behaviour, a certain kind of pattern can be noticed.

There have been more test adding other features like humidity, wind speed, among others but the results have not been improved at all.

Note that all the error percentages presented in references 3.5.2.6 and 3.5.2.7 have not been transformed, again, to the proper scale.

4.5.3. R Stuttgart Neural Network Simulator library

In order to improve the results, once an exhaustive analysis of the variables has been carried out, a new R package is used for the analysis.

The chosen package is the denominated R Stuttgart Network Simulator (RSNNS), which consists in one of the software's that support the highest number of models (multi-layer perceptrons, recurrent Elman, Jordan networks, radial basis function networks among others). SNNS (Stuttgart Neural Network Simulator) is a simulator for neural networks developed at the Institute for Parallel and Distributed High Performance Systems (Institut für Parallele und Verteilte Höchstleistungsrechner, IPVR) at the University of Stuttgart since 1989.

Besides the huge number of models that can be applied, focusing on the multi-layer perceptron (mlp) possibilities, it offers a wide range of learning, update and hidden activation functions. Thanks to this variety, it allows us to seek for better results.

As in the other packages, one of the fundamental pieces of the model are the units: input, hidden and output units.

The SNNS default activation function is Act_logistic which computes the network input simply by summing over all weighted activations and then squashing the result with the logistic function ($f_{act} = \frac{1}{(1+e^{-x})}$). The output function computes the output of every unit from the current activation of this unit.

4.5.3.1. Learning in RSNNS

4.5.3.1.1 General learning procedure

An important spot to focus on with the neural network research, is the question of how to adjust the weights of the links to get the desired system behaviour. This modification is often based on Hebbian rule. It can put in the general form as: $\Delta w_{ij} = g\big(a_j(t), t_j\big)h(o_i(t), w_{ij})$ where:

w_{ij}: weight of the link from unit i to unit j.

a_j: activation of unit j in step t.

t_j: teaching input, desired output of unit j.

$o_i(t)$: output of unit i at time t.

$g(\ldots)$: function, depending on the activation of the unit and the teaching input.

$h(\ldots)$: function, depending on the output of the preceding element.

In this project the training is carried on using, first, a feed-forward neural network, and then with a Backpropagation model, both, with supervised learning. This techniques consists in the following procedure:

An input pattern is presented to the network. The input is then propagated forward in the net until activation reaches the output layer (forward propagation phase). The output of the output layer is then compared with the teaching input giving the error, or difference, δ_j, between the output o_j and the teaching input t_j of a target input unit j is then used together with the output o_i of the source unit i to compute the necessary changes to the link w_{ij}.

To compute the deltas of inner units for which no teaching input is available, (units of hidden layers) the deltas of the following layer, which are already computed, are used in a formula given below. In this way the errors (deltas) are propagated backward, so this phase is called backward propagation.

In this case study offline learning, or batch learning, is performed so the weight changes are cumulated for all patterns in the training set and the sum of all changes is applied after one full cycle (epoch).

4.5.3.1.2 Backpropagation learning functions

The most famous learning algorithm which works in the manner described is currently backpropagation. In the backpropagation learning algorithm online training is usually significantly faster than batch training, especially in the case of large training sets with many similar training examples.

The backpropagation weight update rule, also called generalized delta-rule reads as follows:

$$w_{ij} = \eta \delta_j o_i$$

$$\delta_j \begin{cases} f_j(net_j)(t_j - o_j) \; if \; unit \; j \; is \; an \; output - unit \\ f_j(net_j) \sum_k \delta_k w_{jk} \; if \; unit \; j \; is \; a \; hidden - unit \end{cases}$$

η: learning factor eta (constant).
δ_j: error (difference between the real output and the teaching input) of unit j.
t_j: teaching input of unit j.
o_i: output of the preceding unit i.
i: index of a predecessor to the current unit j with link $w_{jk}$from i to j.
j: index of the current unit.
k: index of a successor to the current unit j with link $w_{jk}$from j to k.

Another good reason to work with the wrap for R of SNNS library is that there are several options to work with the backpropagation algorithm:

- Std_Backpropagation: also denominated Vanilla Backpropagation is the most common learning algorithm and its formula is shown above.
- BackpropMomentum: this represents and enhancement of the previous algorithm. The momentum term introduces the old weight change as a parameter for the computation of the new weight change. This avoids oscillation problems common with the regular backpropagation algorithm when the error surface has a very narrow minimum area. The new weight change is computed by: $\Delta w_{ij}(t+1) = \eta \cdot \delta_j \cdot o_i + \alpha \cdot \Delta w_{ij}(t)$ where α

is a constant specifying the influence of the momentum. The effect of these enhancements is that flat spots of the error surface are traversed relatively rapidly with a few big steps, while the step size is decreased as the surface gets rougher. This adaption of the step size increases learning speed significantly.

- BackpropBatch: Batch backpropagation has a similar formula as vanilla backpropagation. The difference lies in the time when the update of the links takes place. While in vanilla backpropagation an update step is performed after each single pattern, in batch backpropagation all weight changes are summed over a full presentation of all training patterns (one epoch). Only then, an update with the accumulated weight changes is performed. This update behavior is especially well suited for training pattern parallel implementations where communication costs are critical.
- BackpropWeightDecay: It decreases the weights of the links while training them with backpropagation. In addition to each update of a weight by backpropagation, the weight is decreased by a part d of its old value: $\Delta w_{ij}(t+1) = \eta \cdot \delta_j \cdot o_i - d \cdot \Delta w_{ij}(t)$. Weight decay is simply a tendency for weights to be reduced very slightly every time they are updated.

4.5.3.1.3 Quickpropagation learning function

Another method, used to speed up the learning process, is the denominated Quickprop. It uses the information about the curvature of the error surface which requires the computation of second order derivatives of the error function. Quickprop assumes the error surface to be locally quadratic and attempts to jump in one step from the current position directly into the minimum of the parabola.

Quickprop computes the derivatives in the direction of each weight. After computing the first gradient with regular backpropagation, a direct step to the error minimum is attempted by: $\Delta(t+1)w_{ij} = \frac{S(t+1)}{S(t)-S(t+1)} \cdot \Delta(t)w_{ij}$ where:

w_{ij}: weights between units i and j.

$\Delta(t+1)$: actual weight change.

$S(t+1)$: partial derivate of the error function by w_{ij}.

$S(t)$: the last partial derivate.

4.5.3.1.4 Resilient propagation learning function

Last but not least, it is a must to introduce the Rprop learning function. Rprop stands for ' Resilient back propagation' and is a local adaptive learning scheme, performing supervised batch learning in multi-layer perceptrons.

The basic principle of Rprop is to eliminate the harmful influence of the size of the partial derivative on the weight step. As a consequence, only the sign of the derivative is considered to indicate the direction of the weight update. The size of the weight change is exclusively determined by a weight-specific, so-called 'update-value'$\Delta_{ij}^{(t)}$.

$$\Delta w_{ij}^{(t)} \begin{cases} -\Delta_{ij}^{(t)}, if \ \frac{\partial E^{(t)}}{\partial w_{ij}} > 0 \\ +\Delta_{ij}^{(t)}, if \ \frac{\partial E^{(t)}}{\partial w_{ij}} < 0 \\ 0, else \end{cases}$$

Where $\frac{\partial E^{(t)}}{\partial w_{ij}}$ denotes the summed gradient information over all patterns of the pattern set ('batch learning').

The second step of Rprop learning is to determine the new update-values. This is based on a sign-dependent adaptation process.

$$\Delta_{ij}^{(t)} \begin{cases} \eta^{+} \cdot \Delta_{ij}^{(t-1)}, if \ \frac{\partial E^{(t-1)}}{\partial w_{ij}} \cdot \frac{\partial E^{(t)}}{\partial w_{ij}} > 0 \\ \eta^{-} \cdot \Delta_{ij}^{(t-1)}, if \ \frac{\partial E^{(t-1)}}{\partial w_{ij}} \cdot \frac{\partial E^{(t)}}{\partial w_{ij}} < 0 \\ \Delta_{ij}^{(t-1)}, else \end{cases}$$ Where $0<\eta^{-} < 1 < \eta^{+}$.

For Rprop tries to adapt its learning process to the topology of the error function, it follows the principle of 'batch learning' or 'learning by epoch'. That means, that weight-update and adaptation are performed after the gradient information of the whole pattern set is computed.

The Rprop algorithm takes three parameters: the initial update-valueΔ_0, a limit for the maximum step size, Δ_{max}, and the weight-decay exponent α.

One of the major advantages of neural nets is their ability to generalize. This means that a trained net could classify data from the same class as the learning data that it has never seen before. In real world applications developers normally have only a small part of all possible patterns for the generation of a neural net. To reach the best generalization, the dataset should be split into three parts:

- The training set is used to train a neural net. The error of this dataset is minimized during training.
- The validation set is used to determine the performance of a neural network on patterns that are not trained during learning.
- A test set for finally checking the overall performance of a neural net.

4.5.3.2. Update Functions in RSNNS

The update function is an important concept as it is necessary to visit the neurons of a net in a specific sequential order to perform operations on them. For this particular case study there are several update functions that can be used:

- Serial_Order: calculates the activation and output value for each unit. The progression of the neurons is serial which means the computation process starts at the first unit and proceeds to the last one.

- Synchronous_Order: all neurons change their value at the same time. All neurons calculate their activation in one single step. The output of all neurons will be calculated after the activation step. Very useful for distributed systems (SIMD).
- Topological_Order: This mode is the most favourable mode for feedforward nets. The neurons calculate their new activation in a topological order. The topological order is given by the net-topology. This means that the first processed layer is the input layer. The next processed layer is the first hidden layer and the last layer is the output layer. A learning cycle is defined as a pass through all neurons of the net. Shortcut-connections are allowed.

4.5.3.3. Initialization Functions in RSNNS

In order to work with several neural network models and learning algorithms, different initialization functions that initialize the components of a net are required. Backpropagation, for example, will not work properly if all weights are initialized to the same value.

The initialization function that fits better the model are the following one:

- Randomize_Weights: This function initializes all weights and the bias with distributed random values. The values are chosen from the interval$[\alpha;\beta]$. It is required that $\alpha > \beta$.It is important to know that if the seed is not changed in each test, those random values are going to be the same as default.

4.5.3.4. Algorithm development

Finally, due to the wide range of features that the RSNNS allows to work with, this library is chosen to develop the final solution for the project. Also it is important to keep in mind that the new code is going to be developed and tested using, specifically, the respiratory type dataset.

The explicative variables from the dataset are not chosen again, but they are tested all over again. In order to carry out the study, using a simple version of the neural network, the ARV (average relative variance) and MSE (mean squared error) are calculated in order to prove the accuracy of the predictions. The MSE is widely used in statistics as an estimator of the average of the squared errors, which corresponds to the difference between the estimator and what is estimated. The MSE has been calculated as$\sqrt{mean(observations - predictions)^2}$. When working with predictive models, another useful indicator for sizing the goodness of the model is the denominated ARV which is calculated from dividing the MSE per the variance of the dataset: $\frac{\sum(observations-predictions)^2}{\sum(observations-mean(observations))^2}$. As the output of the data set has been divided by 50, in order to scale the output within 0 and 1, all the values calculated with both expressions, are going to be affected.

4.5.3.4.1.1 Variable introduction

Step by step analysis is going to be performed, adding gradually each one of the explicative variables that were previously discussed. This study should help to understand better the effect of each variable when is introduced into the model. Once all the variables are applied together it would be useful to perform an interaction analysis between them.

	$A_0 \mid T_0$	$A_0\, T_1\, T_2\, T_3\, T_7 \mid T_0$
ARV	0,348	0,250
MSE	0,090	0,076

Table 3. ARV and MSE for two different data sets

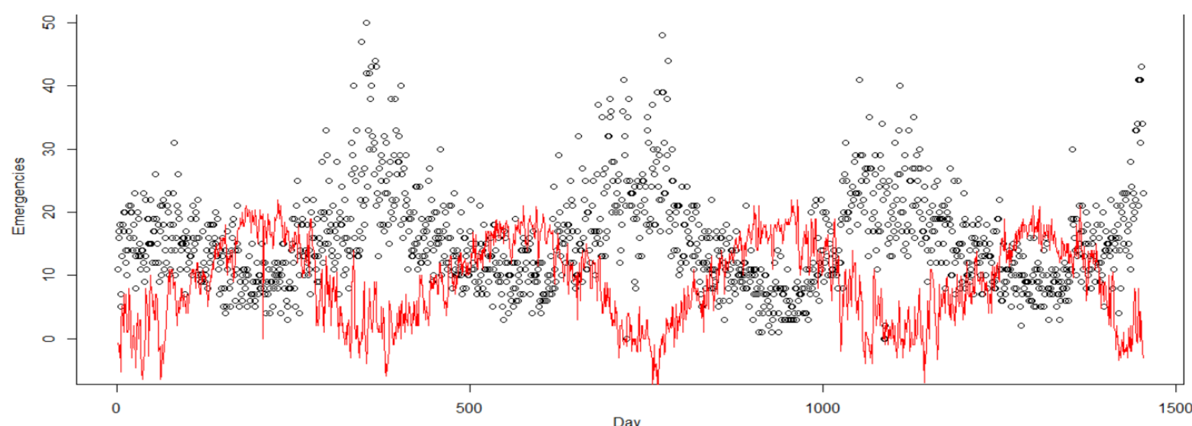
Where:

A_0 is the number of emergencies in the morning (from 07:00 to 13:00).

T_x is the total number of emergencies with x delays.

The results lead to think that the neural network is capable to capture the process dynamics as it is proved that adding the information of the own series with delay help to improve the performance. In order to enhance the results, it seems that more explicative variables should be added to the model. Several temperatures (T_{max}, T_{min}, T, and some temperature indicators) have been tested without obtaining good results. For keep improving the model a study of the errors have been performed.

In order to justify the addition of the minimum temperature as a variable in the data set, it is useful to observe the plot below.



Plot 29. Number of respiratory emergencies (red) and minimum temperature of the day per day (2010-2014)

One of the reasons to work with the neural networks is to be able to spot values that would be treated as outliers in many other studies. The temperature time series, should help the algorithm to find this particular cases. Is easy to appreciate how those graphs have opposite behaviours that would let refine the predictions. This can be observed by calculating the correlation between both time series, which has a value of -0,546.

4.5.3.4.1.2 Error analysis

First step consists in obtain the values for the error between the predictions and the real data observed in order to carry out a temporal series analysis with the result.

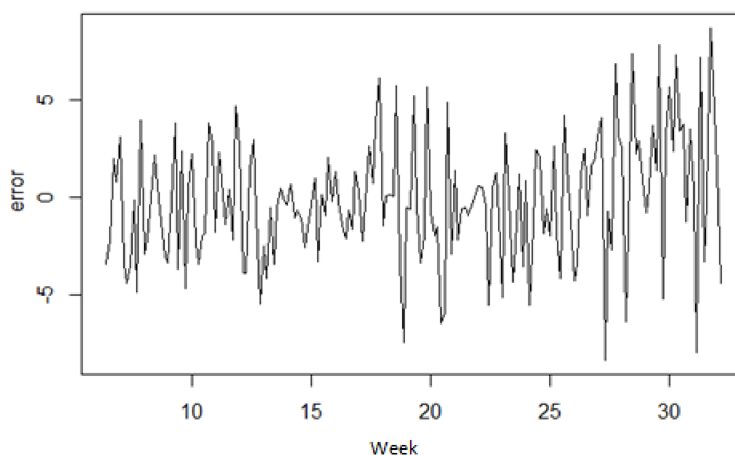
```
error <- (observations-predictions)

error <- ts(error, start=c(6,4), freq=7)
```

ETSEIB

The temporal series has a frequency equals to seven (days in a week) and starts on Thursday (4) of June (6). The time axis, used in all the plots below, corresponds to the last 30 weeks of the year 2013 which is the one that is being used as a validation set.

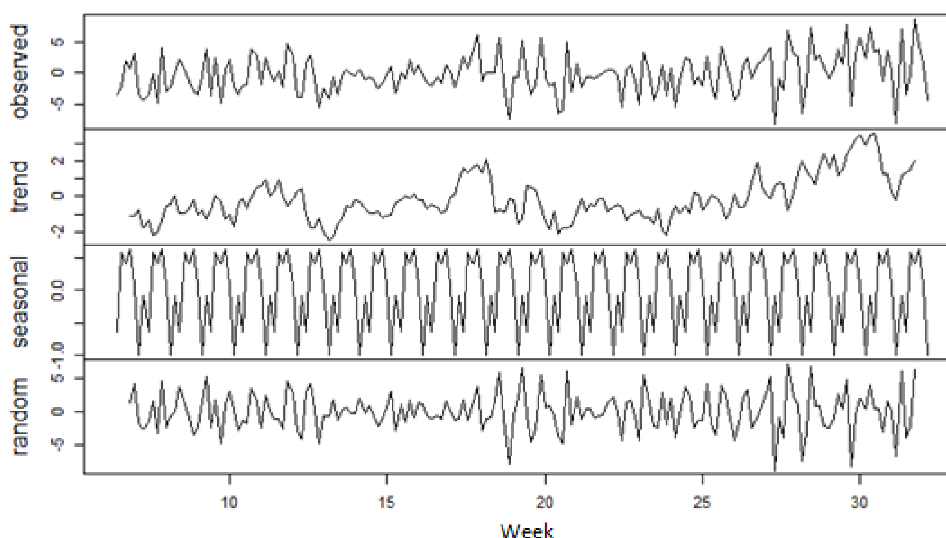
Setting the frequency to seven, is going to set a time of study of a 25 weeks and two days, which are the ones of the last 6 months of year 2013.

First is interesting to observe the plot of the error in order to see how does it looks like.



Plot 30. Prediction's error plot

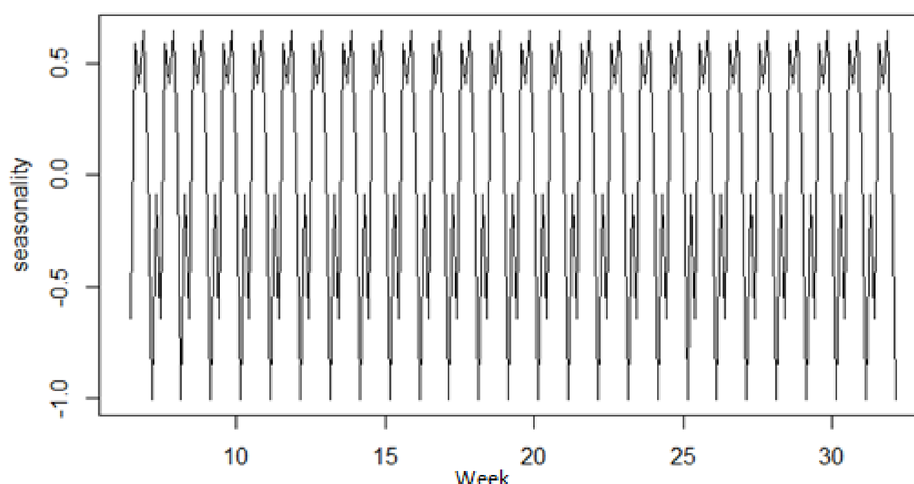
The error is distributed around the zero value, but it increases for the last predictions, which are mainly, for the month of December (the number of respiratory type emergencies tend to i ncrease in this period). For further investigations, plot (decompose(error)) is being applied, o btaining the following plots:



Plot 31. Analysis of the error time series

From down to the top, the random plot presents a random distribution of the error, the seaso nal graph shows a seasonality component that should be investigated.
The trend plot strength the argument that there exists an undeniable growth of the error in th e last period of the analysis which also correspond to a highest registers of emergencies.

Seasonal component



Plot 32. Seasonal plot of the error

Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
0,09	-1,10	-0,05	-0,5	0,24	0,49	0,63

Table 4. Values of the seasonal component

Tuesday has the higher error among the week followed by Saturday and Sunday. From the descriptive analysis, done during the first part of the project, was observed that the higher frequency of emergencies during the week were on the weekend. According to the previous conclusion a new variable, has been introduce in the data set: Day of the week (Dow) indicating with a 1 the days within the weekend and with 0 the others.

Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
0,27	-0,88	0,14	-0,44	0,38	0,11	0,39

Table 5. Values of the seasonal component

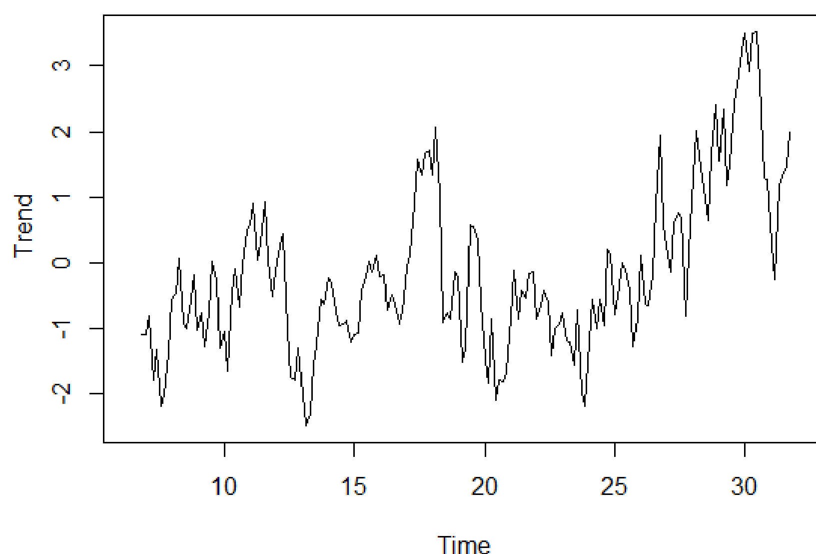
Thanks to the new variable the seasonal component has been considerably reduced. Finally, in order to decrease Tuesday's parameter, this day has been indicated with a -1 obtaining:

Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
0,17	-0,34	0,09	-0,38	0,50	-0,13	0,10

Table 6. Values of the seasonal component

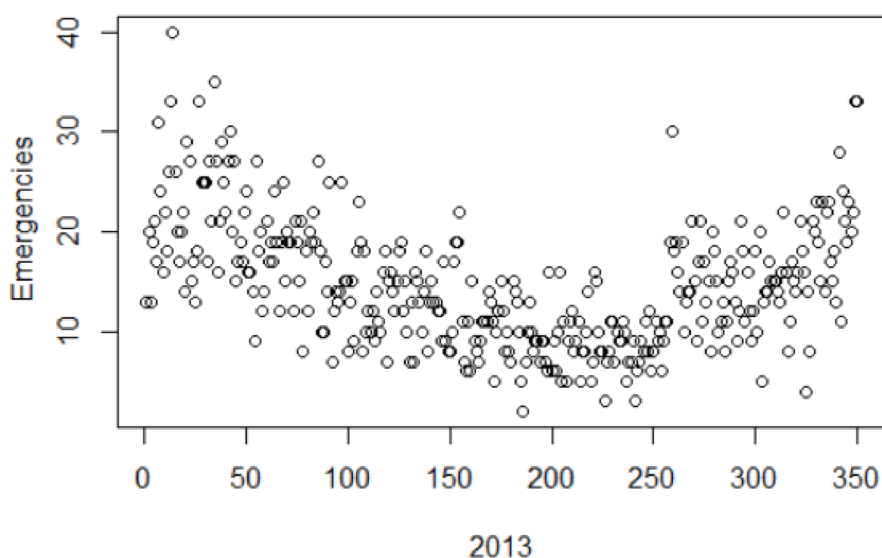
Once the seasonal component of the error time series has been allocated and reduced, and in order to improve the final prediction, the trend component should be also lowered.

Trend component



Plot 33. Trend plot of the error

As indicated above, the trend clearly increases for the last months, especially on December, of 2013. One interesting thing to notice is that the trend of the error is increasing when the total number of emergencies increases during the year. This can be observed by showing the graph for the total emergencies in 2013.



Plot 34. Respiratory emergencies in year 2013

One solution to capture this behaviour would be to add a dummy variable to indicate if the month should be analysed as high frequency period (winter and autumn) indicated with a 1, or low frequency (spring and summer) designated with a 0. Despite the introduction of this new dummy variable to the model, it is still not able to capture this phenomenon and cannot lower the trend for this specific periods. In order to apply a more particular solution, the

month of December, which is the one where the trend substantially grows, has been designated with a 1 while keeping the other months as 0 and still no differences.

In order to lower the trend, several variables have been tested, dummies and others like temperature, or indexes combining temperature and humidity information. Any of them has worked as supposed to. Knowing that, and for improving the model, the neural network parameters and functions should be as tuned as possible.

For solving this situation a part of code has been added to the algorithm in order to visualize which the days with the higher errors are during the training and testing.

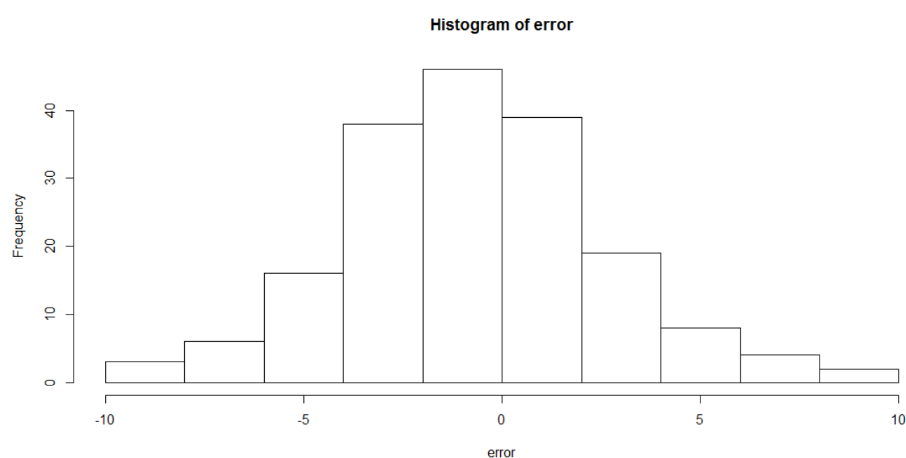
It is also important to know which the biggest errors are done during the training of the neural network. The top 20 errors have been collected thanks to a matrix that is populated during the training period. Tested several times, the values in the top 10 tend to be shown every time.

ID	A1.0	T0	Date	Weekday	Month
1066	7	35	08/12/2012	6	12
698	9	38	06/12/2011	2	12
81	6	31	29/03/2010	1	3
751	6	33	28/01/2012	6	1
723	0	0	31/12/2011	6	12
8	0	18	15/01/2010	5	1
502	3	22	24/05/2011	2	5
688	10	35	26/11/2011	6	11
365	4	31	07/01/2011	5	1
311	5	28	14/11/2010	7	11
1087	0	0	29/12/2012	6	12
654	9	32	23/10/2011	7	10
725	13	38	02/01/2012	1	1
350	4	27	23/12/2010	4	12
1122	9	33	02/02/2013	6	2
442	4	25	25/03/2011	5	3
637	2	21	06/10/2011	4	10
699	8	35	07/12/2011	3	12
737	1	19	14/01/2012	6	1
1133	7	29	13/02/2013	3	2

Table 7. Table of the highest errors obtained in the training

From the top error table, seems that the days with high frequency of emergencies (T0) are not well predicted. Also there are two days were the total entries are 0 which probably are real errors in the data set. As observed in the error time series study, the days with the highest number of respiratory emergencies, are the ones within the “cold” months that were indicated, without success, with a 1 (dummy variable). This table gives real valuable information in order to improve the model as confirms the need to introduce a variable capable of capture the dynamics for this particular months.

The histogram of the residuals shows that the variance is normally distributed. In the plot above, a bell-shaped histogram which is evenly distributed around zero is shown.



Plot 35. Histogram of the error

4.5.3.4.1.3 Cross Validation

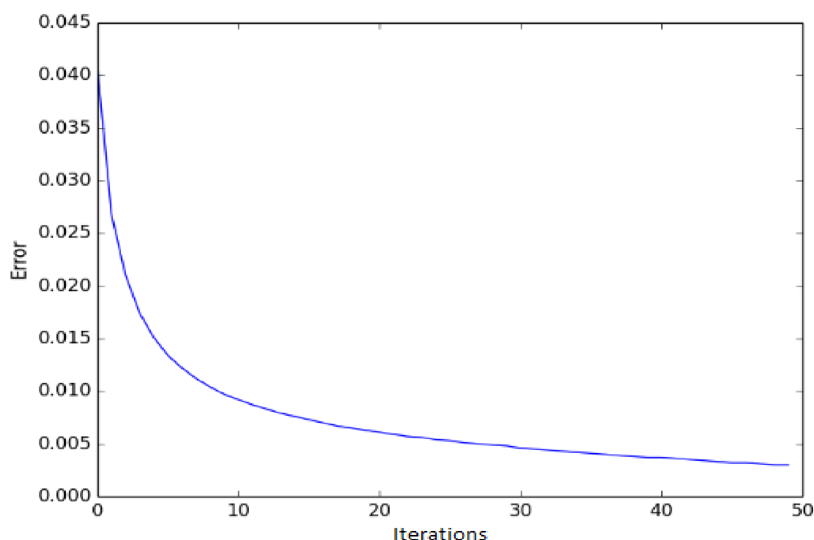
Cross Validation is going to be used as a model validation technique for assessing how the results of a statistical analysis will generalize to an independent data set. This also will allow to know how accurately the model is performing in practice. There are several types of cross validation and they can be grouped as exhaustive and non-exhaustive types.

Using an exhaustive method, the model will learn and test in all the possible ways to divide the original sample into a training and validation set. On the other hand, the non-exhaustive methods, do not compute all ways of splitting the original sample.

For this project is chosen to use the k-fold cross validation, which is one of the non-exhaustive methods. In k-fold, the original sample is randomly partitioned into k equal sized subsamples. Of the k samples, a single subsample is retained as the validation data for testing the model, and the remaining k-1 subsamples are used as training data. This methodology has been chosen according to several good references from similar projects developed by some universities.

4.5.3.4.2 Consistency analysis

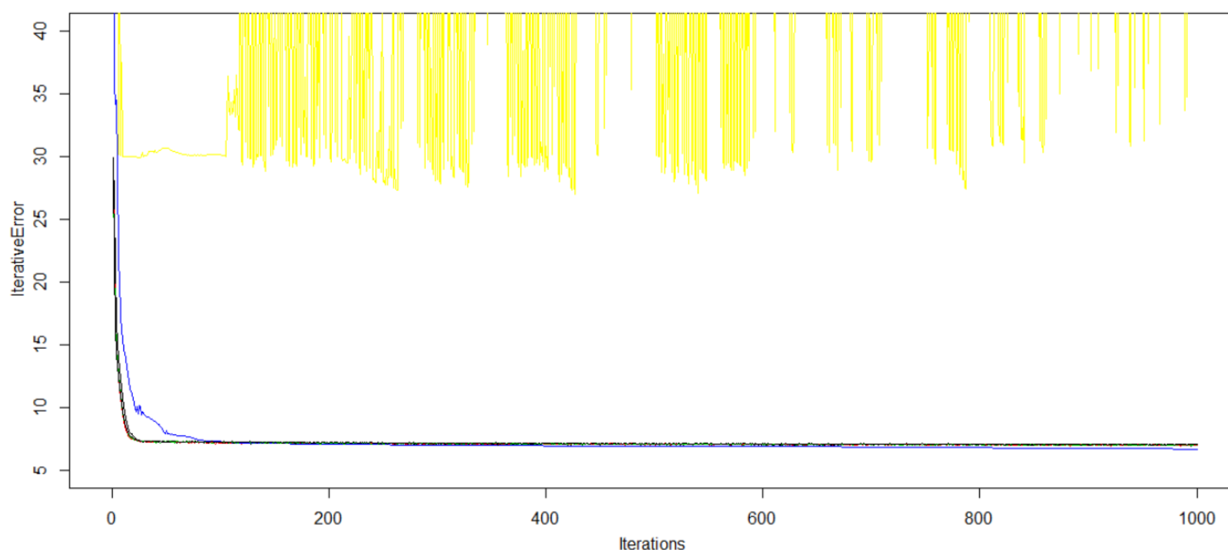
A key indicator that help to improve the results of the algorithm and that is widely related with some principal neural network parameters such as the learning function or the size, is the error curve generated during the training. This curve, usually has a hyperbolic shape such as the one shown in the plot below.



Plot 36. Training error for each iteration.

Several conclusions can be obtained from this curve, such as how many iterations took, during the training, to reach a constant error (or a minimum spot) or even can discover if the algorithm is over fitting the weights.

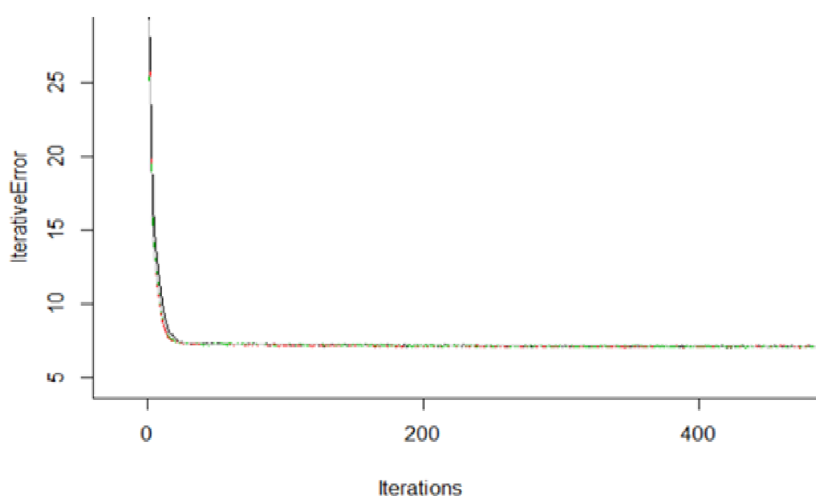
In order to determine the different shapes that this error curve for the data set, several trainings are performed only changing the learning function for each case. All the results are compared in the same plot.



Plot 37. Error in the training for each algorithm iteration and learning function

This plot is displaying the iterative error for each iteration for the following learning functions (with a size of ten and a logistic output as settings): *BackpropWeightDecay* (black), *Std_Backpropagation* (red), *Rprop* (blue), *Quickprop* (yellow), *BackpropMomentum* (green) and *BackpropChunk* (grey).

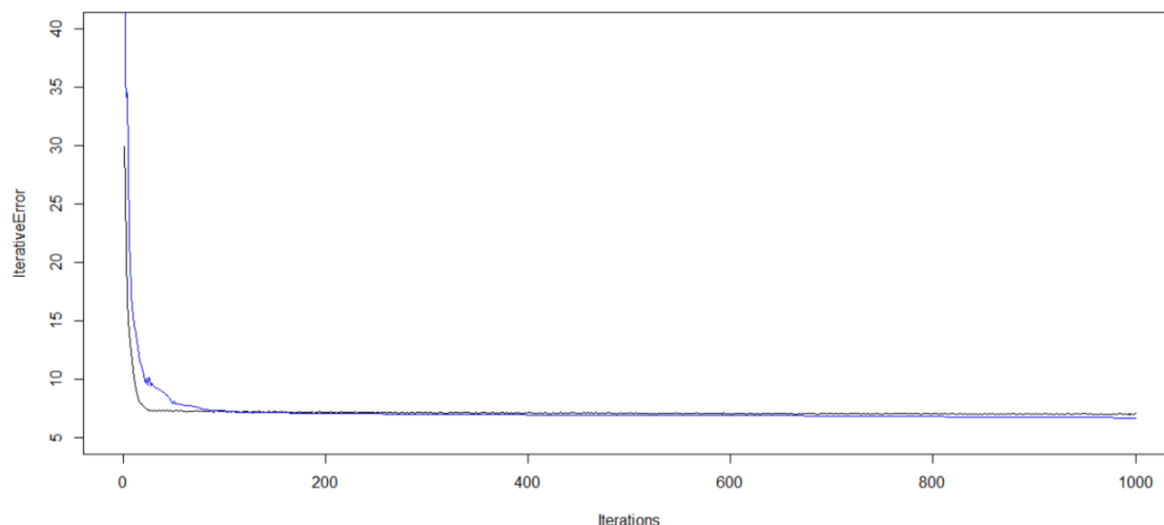
This plot clearly shows that for this data set, and despite using different parameters to try to tune the algorithm, the *Quickprop* learning function has an unpredictable error function which is better to avoid while the rest of the models are showing a similar behaviour.



Plot 38. Error in the training for each algorithm iteration and learning function

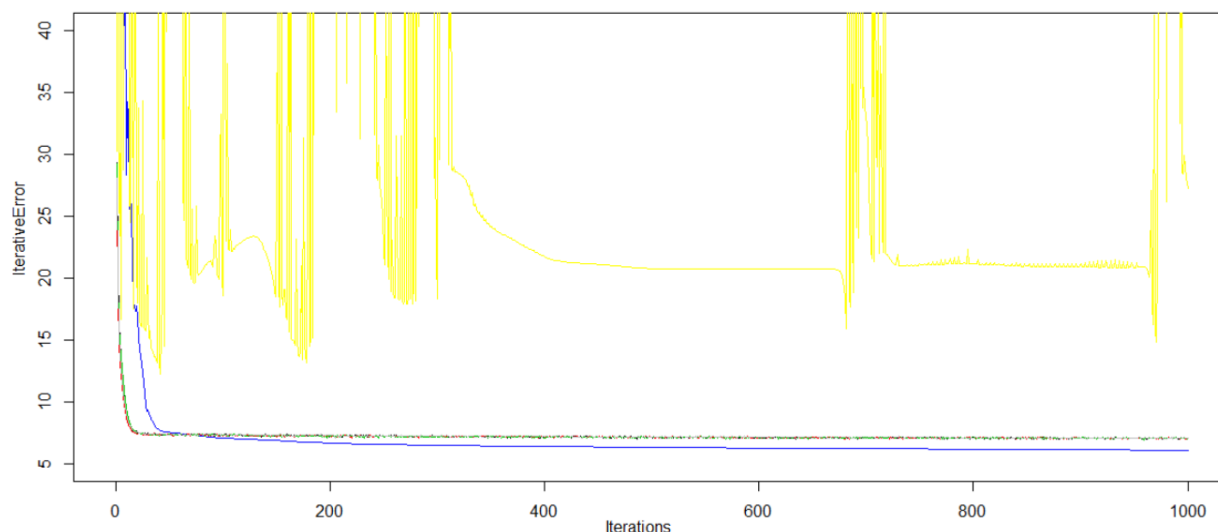
Showing the plot for the backpropagation functions (the *Rprop* has been keep out of the graph) discovers that all of them have a similar shape and they tend to approximately the same value at the same number of iterations.

In the other hand, there is the *Rprop* learning function which decreases slower than the backpropagation functions, and reaches a lower level of error when the computation has finished. This interpretation could led towards a bad conclusion, as how it is saw in the factorial experiment, the *Rprop* perform better during the training, but due a possible over-fitting, the results in test are worse than using the other learning functions.



Plot 39. Error in the training for each algorithm iteration and learning function

Another plot with the same error curves obtain from a multi-layer perceptron with a size of 25 instead of 10. In this one, the *Quickprop* (yellow) learning function still working bad, and *Rprop* has the same previous behaviour but a little bit more accentuated.



Plot 40. Error in the training for each algorithm iteration and learning function

4.5.3.4.2.1 Variable normalization

According to multiple tests, the normalization of the variables from the data set is very important to work with the neural networks. The standard procedure consists in transform each variable in order to have it within 0 and 1. This is because when using the logistic

activation function, the values should be set as described. Without applying this normalization the neural network is not working properly.

There are different methodologies to transform the data set variables into a range between 0 and 1. The easiest and most used ones, are the standard normalization $\frac{x-mean(x)}{var(x)}$ or, even faster, divide the data set by the maximum value. Despite those methods work accurately and allow us to work with the logistic activation function, sometimes, they could be covering up or be distorting some valuable intrinsic information from the variable inner distribution. This is why, as a previous analysis, is worthy to perform an study of which one of the distributions (Normal, Poisson, Exponential, Weibull, etc.) fit better each variable in order to apply the right normalizations respectively. In this case any of the distributions is fitting the variables, hence the standard normalization is choose.

4.5.3.5. Topology selection

4.5.3.5.1.1 Factorial experiment

As there are several features that can be used in the RSNNS library, the most important and the ones that have been analysed in the RSNNS section are chosen to be part of a factorial experiment. This factorial experiment is going to have the following independent variables and levels:

- Learning function: *Std_Backpropagation, BackpropBatch, BackpropChunk, BackpropMomentum, BackpropWeightdecay, Rprop.*
- Size (number of units in the hidden layer): low (5), moderate (20), high (50) and very high (100).
- linOut: sets the activation function of the output units to linear or logistic (Boolean values).

The dependent variables are ARV and MSE, but they are calculated two times: one as the result of the training and another for the testing. This is important, as it has been noticed that some good features for training offers worst results in testing than the others. So it is recommendable to perform this kind of analysis when working with neural networks.

Another feature that should be monitored is the proportion of the data set that is used for training and testing. All the analysis previously carried out consists in, by using three years and a half, predict the last half of the forth month. Perform the analysis with other proportions (75%-25% or even 50% - 50%) could clarify how the neural network is calculating the results.

First Factorial Experiment: using three years and a half for training and half a year for testing.

The results are written in a csv file. Thanks to a dynamic table several conclusions can be assumed. The analysis is going to be perform by little by little adding new dimensions (explicative variables). For all the tables displayed in the following analysis the values MSE1 and ARV1 are the results for the training of the neural network (fitting measure) and MSE2 and ARV2 for the test (prediction measure).

First, the performance of the different learning functions is tested for both training and testing sets. It is important to remark that each learning function parameters are initialized, in the neural network, with its default values

	MSE1	ARV1	MSE2	ARV2
BackpropBatch	0.1614	1.7139	0.1543	1.6821
BackpropChunk	0.0748	0.2364	0.0631	0.1856
BackpropMomentum	0.0745	0.2345	0.0629	0.1845
BackpropWeightDecay	0.0745	0.2348	0.0626	0.1830
Rprop	0.0696	0.2059	0.0662	0.2059
Std_Backpropagation	0.0745	0.2344	0.0629	0.1850

Table 8. ARV and MSE values for each Learning function

The initial results show that all the values get better for test, but this is due there are less values. What is remarkable from the table are the following conclusions:

- The *BackpropBatch* does not work properly for this data set.
- The other learning functions based on the backpropagation algorithm are working quite similar.
- The *Rprop* learning function provide the best values in train but the worst in test. This phenomenon happens due to a possible over fitting of the model.

Next step, two levels (True or false) of the categorical variable l*inOut* haven been added to the table. It does not bring new important conclusions as the values are similar.

	MSE1	ARV1	MSE2	ARV2
BackpropBatch				
FALSE	0.1321	0.7462	0.1302	0.8005
TRUE	0.1908	2.6816	0.1782	2.5638
BackpropChunk				
FALSE	0.0743	0.2336	0.0623	0.1811
TRUE	0.0751	0.2391	0.0638	0.1901
BackpropMomentum				
FALSE	0.0747	0.2361	0.0626	0.1829
TRUE	0.0742	0.2330	0.0632	0.1861
BackpropWeightDecay				
FALSE	0.0744	0.2339	0.0624	0.1815
TRUE	0.0746	0.2353	0.0629	0.1846
Rprop				
FALSE	0.0691	0.2025	0.0672	0.2121
TRUE	0.0703	0.2093	0.0652	0.1997
Std_Backpropagation				
FALSE	0.0746	0.2358	0.0628	0.1841
TRUE	0.0742	0.2330	0.0631	0.1858

Table 9. ARV and MSE values for each Learning function and each output function combination

As all the explicative variables are set to be within 0 and 1, the l*inOut* parameter is set to false as it implies that the output is not linear but logistic.

Next step consists in analyse the results with different sizes of the hidden layer. This is a tricky decision as there are several external factors implied, when looking for the best number of hidden units. This factors are the following ones according many documentation:

Number of input and output units, number of training cases, amount of noise in the targets, the complexity of the function or classification to be learned, the architecture, the type of hidden unit activation function, the training algorithm and regularization.

In most situations, there is not a specific methodology to determine the best number of hidden units without training several networks and estimating the generalization error of each. If there are too few hidden units, there will be high training error and high generalization error due to under fitting and high statistical bias. If there are too many hidden units, there will be low training error but still have high generalization error due to over fitting and high variance. There are lots of solutions regarding this problem but most of them only focuses in providing solutions for only some of the factors, but do not offer a good general solution.

The best recommendation is to perform a sensitive analysis, similar to the one that is carried out in this project, in order to see for which size of the neural network the generalization error is the lowest one. This generalization error, in machine learning, consists in measure how well a learning machine generalizes to unseen data. It is measured as the distance between the error on the training set and the test set and is averaged over the entire set of possible training data that can be generated after each iteration of the learning process. As this parameter is not easy to obtain, the goodness of each model is related to MSE and ARV values.

	MSE1	ARV1	MSE2	ARV2
BackpropBatch				
2	0.1319	0.7475	0.1299	0.8019
10	0.1215	0.6445	0.1178	0.6738
50	0.1077	0.5026	0.1030	0.5141
100	0.2847	4.9612	0.2662	4.7386
BackpropChunk				
2	0.0748	0.2369	0.0627	0.1841
10	0.0740	0.2320	0.0628	0.1841
50	0.0745	0.2348	0.0632	0.1868
100	0.0755	0.2416	0.0633	0.1871
BackpropMomentum				
2	0.0748	0.2371	0.0624	0.1831
10	0.0738	0.2307	0.0626	0.1831
50	0.0745	0.2348	0.0631	0.1863
100	0.0746	0.2353	0.0630	0.1853
BackpropWeightDecay				
2	0.0744	0.2340	0.0620	0.1794
10	0.0736	0.2294	0.0625	0.1826
50	0.0744	0.2344	0.0630	0.1854
100	0.0754	0.2403	0.0629	0.1846
Rprop				
2	0.0740	0.2314	0.0612	0.1748
10	0.0720	0.2194	0.0628	0.1840
50	0.0684	0.1982	0.0676	0.2139
100	0.0642	0.1746	0.0732	0.2506
Std_Backpropagation				
2	0.0742	0.2330	0.0617	0.1779
10	0.0743	0.2335	0.0635	0.1885
50	0.0741	0.2325	0.0629	0.1851
100	0.0750	0.23845	0.06350	0.18827

Table 10. ARV and MSE values for each Learning function and for a layer size of 2, 10, 50 and 100 units

The table shows how increasing the size from two to ten units the results seem to improve in most of the cases. Performing the same increase but from ten to fifty the same results are showing quite higher, so not working as good as the past configuration. The minimum is probably between ten and fifty.

Based on the results and conclusions from the table, a model with *BackpropWithDecay* as a learning function with a size between ten and fifty units (finally is decided to work with 25 units) and the l*inOut* set to False.

Second Factorial Experiment: using three years for training and a year for testing.

	MSE1	ARV1	MSE2	ARV2
BackpropBatch	0.1523	1.3783	0.1399	1.2516
BackpropChunk	0.0745	0.2315	0.0679	0.2244
BackpropMomentum	0.0755	0.2379	0.0687	0.2306
BackpropWeightDecay	0.0747	0.2320	0.0679	0.2257
Rprop	0.0688	0.2837	0.0747	0.1923
Std_Backpropagation	0.0750	0.2330	0.0681	0.2273

Table 11. ARV and MSE values for each Learning function

	MSE1	ARV1	MSE2	ARV2
BackpropBatch				
FALSE	0.1340	0.7304	0.1197	0.7351
TRUE	0.1706	2.0261	0.1597	1.7680
BackpropChunk				
FALSE	0.0747	0.2307	0.0677	0.2256
TRUE	0.0743	0.2323	0.0680	0.2232
BackpropMomentum				
FALSE	0.0764	0.2435	0.0695	0.2365
TRUE	0.0746	0.2323	0.0680	0.22477
BackpropWeightDecay				
FALSE	0.0747	0.2307	0.0678	0.2255
TRUE	0.0748	0.2332	0.0681	0.2259
Rprop				
FALSE	0.0677	0.2897	0.075	0.1869
TRUE	0.0698	0.2776	0.073	0.1976
Std_Backpropagation				
FALSE	0.0742	0.2301	0.0677	0.2256
TRUE	0.0753	0.2360	0.0685	0.2291

Table 12. ARV and MSE values for each Learning function and each output function combination

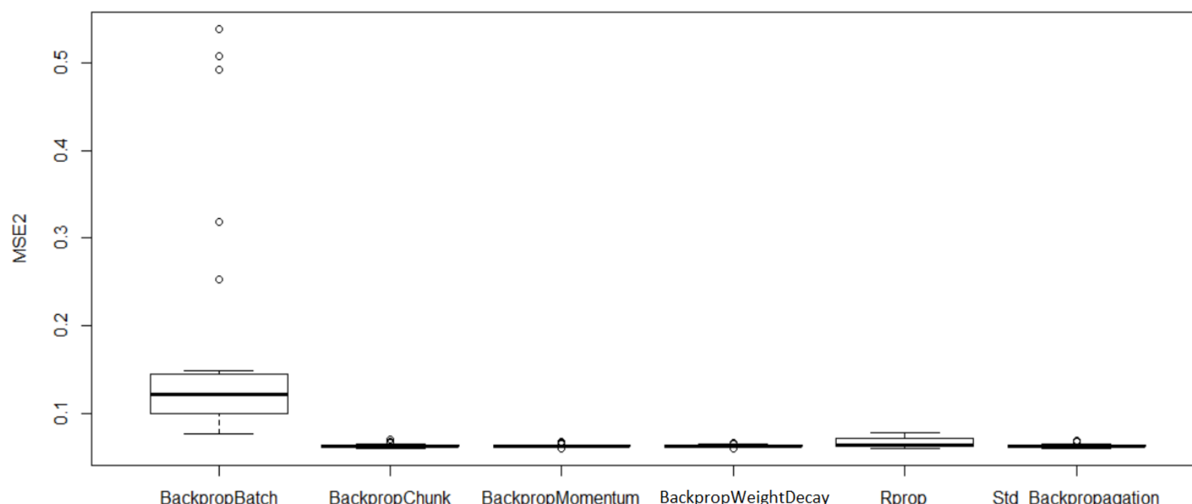
	MSE1	ARV1	MSE2	ARV2
BackpropBatch				
2	0.1333	0.7342	0.1195	0.7350
10	0.1234	0.6340	0.1109	0.6332
50	0.1084	0.4786	0.0964	0.4864
100	0.2441	3.6663	0.2324	3.1516
BackpropChunk				
2	0.0743	0.2240	0.0667	0.2234
10	0.0741	0.2321	0.0679	0.2220
50	0.0747	0.2357	0.0685	0.2254
100	0.07490	0.23433	0.0683	0.2265
BackpropMomentum				
2	0.0747	0.2269	0.0672	0.2255
10	0.0744	0.2312	0.0678	0.2236
50	0.0757	0.2429	0.0695	0.2321
100	0.0772	0.2506	0.0705	0.2413
BackpropWeightDecay				
2	0.0750	0.2280	0.0673	0.2271
10	0.0740	0.23162	0.0679	0.2212
50	0.0746	0.2355	0.0684	0.2249
100	0.0754	0.2327	0.0680	0.2296
Rprop				
2	0.0742	0.2212	0.0663	0.2223
10	0.0720	0.2446	0.0697	0.2096
50	0.0665	0.3162	0.0791	0.1787
100	0.0625	0.3528	0.0836	0.1586
Std_Backpropagation				
2	0.0750	0.2282	0.0674	0.2272
10	0.0748	0.2351	0.0684	0.2264
50	0.0747	0.2356	0.0684	0.2256
100	0.0754	0.2333	0.0681	0.2301

Table 13. Table. ARV and MSE values for each Learning function and for a layer size of 2, 10, 50 and 100 units

The second analysis does not bring new relevant information.

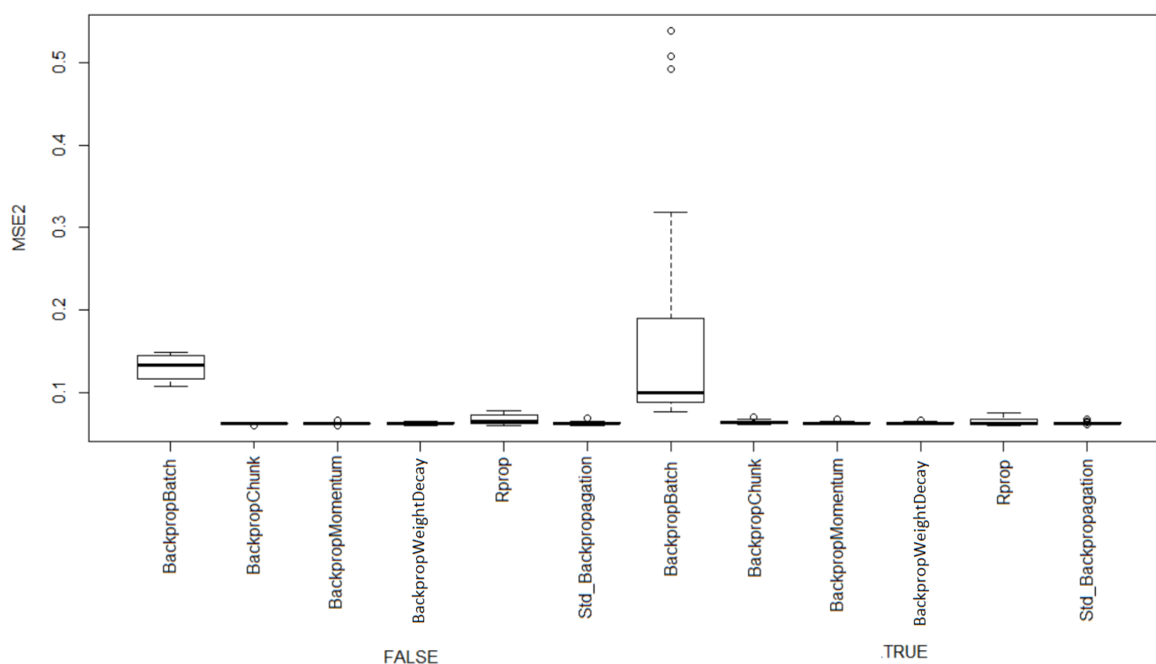
The conclusions from the analysis performed above, can be validated by investigating the error distribution, in the prediction measures (MSE2), for each variable introduced in the factorial experiment. This study can be done by gradually introducing each variable.

First, the MSE2 is evaluated for each learning function. It can be appreciated in the plot, that there is not a difference of the mean for all of them despite in the *BackpropBatch* case. Also, it seems that the *Rprop* values contain a wider range of values than in the other cases.



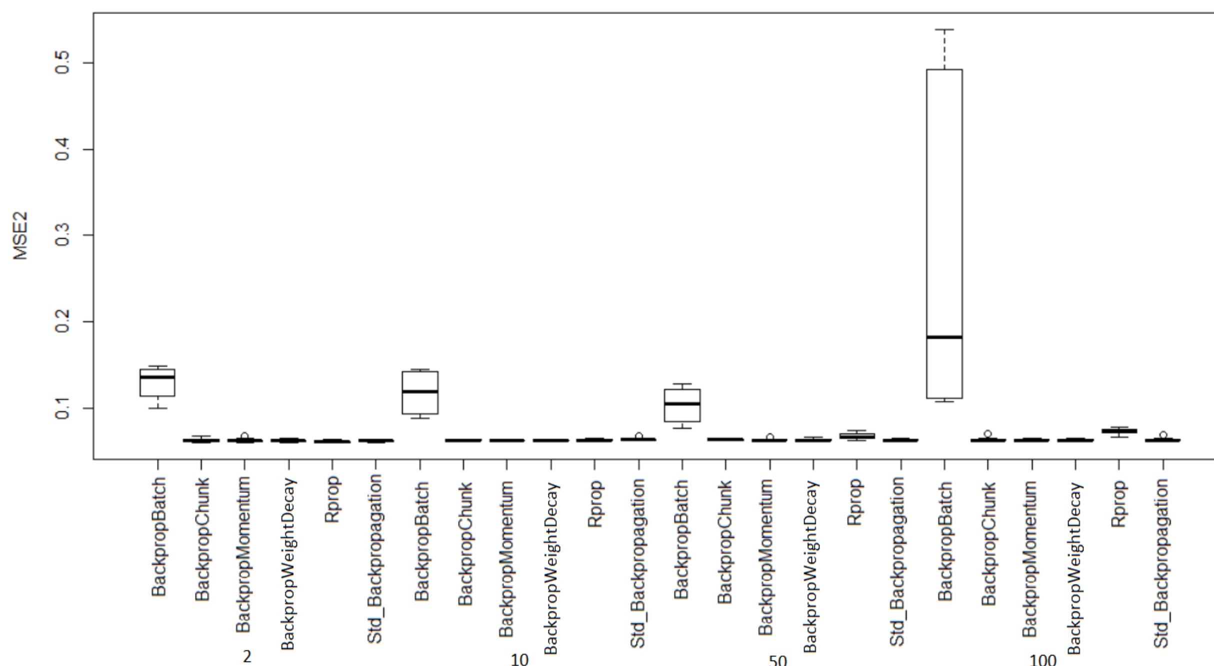
Plot 41. MSE2 vs learning functions

Similarly, there are not significant differences introducing the *linOut* variable which sets the activation function of the output units to linear (TRUE) or logistic (FALSE).



Plot 42. MSE2 vs learning functions and linOut parameter

Evaluating the MSE2 per learning function and size of the layer, the behaviour is similar for sizes 2, 10 and 50, while worse for 100. Comparing 2 and 10, they are in quite the same mean, but 10 hidden units seems to be a better parameter fitting as it shows less variance in the results than in the 2 units case. Finally, it is clear that the measure is increased from 10 to 50.



Plot 43. MSE2 vs learning functions vs number of hidden units

Summary of the model:

Call:
lm(formula = TMSE2 ~ learnF * linear * sizeLay, data = resul)

Residuals:

Min	1Q	Median	3Q	Max
-0.1312	-0.0010	-0.0002	0.0011	0.1771

Coefficients:	
	Estimate
(Intercept)	0.1447
learnFBackpropChunk	-0.0829
learnFBackpropMomentum	-0.0825
learnFBackpropWeightDecay	-0.0830
learnFRprop	-0.0827
learnFStd_Backpropagation	-0.0827
linearTRUE	-0.0912
sizeLay	-0.0003
learnFBackpropChunk:linearTRUE	0.0933
learnFBackpropMomentum:linearTRUE	0.0921
learnFBackpropWeightDecay:linearTRUE	0.0925
learnFRprop:linearTRUE	0.0899
learnFStd_Backpropagation:linearTRUE	0.0924
learnFBackpropChunk:sizeLay	0.0003
learnFBackpropMomentum:sizeLay	0.0003
learnFBackpropWeightDecay:sizeLay	0.0003
learnFRprop:sizeLay	0.0004
learnFStd_Backpropagation:sizeLay	0.0003
linearTRUE:sizeLay	0.0034

learnFBackpropChunk:linearTRUE:sizeLay	-0.0034
learnFBackpropMomentum:linearTRUE:sizeLay	-0.0034
learnFBackpropWeightDecay:linearTRUE:sizeLay	-0.0034
learnFRprop:linearTRUE:sizeLay	-0.0034
learnFStd_Backpropagation:linearTRUE:sizeLay	-0.0034
	Std. Error
(Intercept)	0.0092
learnFBackpropChunk	0.0131
learnFBackpropMomentum	0.0131
learnFBackpropWeightDecay	0.0131
learnFRprop	0.0131
learnFStd_Backpropagation	0.0131
linearTRUE	0.0131
sizeLay	0.0001
learnFBackpropChunk:linearTRUE	0.0185
learnFBackpropMomentum:linearTRUE	0.0185
learnFBackpropWeightDecay:linearTRUE	0.0185
learnFRprop:linearTRUE	0.0185
learnFStd_Backpropagation:linearTRUE	0.0185
learnFBackpropChunk:sizeLay	0.0002
learnFBackpropMomentum:sizeLay	0.0002
learnFBackpropWeightDecay:sizeLay	0.0002
learnFRprop:sizeLay	0.0002
learnFStd_Backpropagation:sizeLay	0.0002
linearTRUE:sizeLay	0.0002
learnFBackpropChunk:linearTRUE:sizeLay	0.0003
learnFBackpropMomentum:linearTRUE:sizeLay	0.0003
learnFBackpropWeightDecay:linearTRUE:sizeLay	0.0003
learnFRprop:linearTRUE:sizeLay	0.0003
learnFStd_Backpropagation:linearTRUE:sizeLay	0.0003
	t value
(Intercept)	15.621
learnFBackpropChunk	-6.33
learnFBackpropMomentum	-6.296
learnFBackpropWeightDecay	-6.338
learnFRprop	-6.313
learnFStd_Backpropagation	-6.316
linearTRUE	-6.964
sizeLay	-2.171
learnFBackpropChunk:linearTRUE	5.036
learnFBackpropMomentum:linearTRUE	4.973
learnFBackpropWeightDecay:linearTRUE	4.993
learnFRprop:linearTRUE	4.853
learnFStd_Backpropagation:linearTRUE	4.986
learnFBackpropChunk:sizeLay	1.589
learnFBackpropMomentum:sizeLay	1.575
learnFBackpropWeightDecay:sizeLay	1.608
learnFRprop:sizeLay	2.086
learnFStd_Backpropagation:sizeLay	1.622
linearTRUE:sizeLay	14.732
learnFBackpropChunk:linearTRUE:sizeLay	-10.459
learnFBackpropMomentum:linearTRUE:sizeLay	-10.445
learnFBackpropWeightDecay:linearTRUE:sizeLay	-10.473

learnFRprop:linearTRUE:sizeLay	-10.467
learnFStd_Backpropagation:linearTRUE:sizeLay	-10.48
	Pr(>|t|)
(Intercept)	2.00E-16
learnFBackpropChunk	1.40E-09
learnFBackpropMomentum	1.68E-09
learnFBackpropWeightDecay	1.34E-09
learnFRprop	1.53E-09
learnFStd_Backpropagation	1.51E-09
linearTRUE	3.93E-11
sizeLay	0.031
learnFBackpropChunk:linearTRUE	1.00E-06
learnFBackpropMomentum:linearTRUE	1.34E-06
learnFBackpropWeightDecay:linearTRUE	1.22E-06
learnFRprop:linearTRUE	2.33E-06
learnFStd_Backpropagation:linearTRUE	1.26E-06
learnFBackpropChunk:sizeLay	0.1136
learnFBackpropMomentum:sizeLay	0.1167
learnFBackpropWeightDecay:sizeLay	0.1094
learnFRprop:sizeLay	0.0382
learnFStd_Backpropagation:sizeLay	0.1063
linearTRUE:sizeLay	<2.00E-16
learnFBackpropChunk:linearTRUE:sizeLay	<2.00E-17
learnFBackpropMomentum:linearTRUE:sizeLay	<2.00E-18
learnFBackpropWeightDecay:linearTRUE:sizeLay	<2.00E-19
learnFRprop:linearTRUE:sizeLay	<2.00E-20
learnFStd_Backpropagation:linearTRUE:sizeLay	<2.00E-21

Table 14. Summary of the factorial experiment

Residual standard error: 0.0287 on 216 degrees of freedom
Multiple R-squared: 0.7682, Adjusted R-squared: 0.7435
F-statistic: 31.12 on 23 and 216 DF, p-value: < 2.2e-16

Finally an ANOVA model can be added to test the relation of the different variables with the output (MSE2):

Response: TMSE2						
	Df	Sum Sq	Mean Sq	F value	Pr(>F)	
learnF	5	0.2746	0.0549	66.6878	< 2.2e-16	***
linear	1	0.0039	0.0039	4.8448	0.0288	*
sizeLay	1	0.0229	0.0229	27.871	3.15E-07	***
learnF:linear	5	0.0191	0.0038	4.6508	0.000472	***
learnF:sizeLay	5	0.0899	0.0179	21.8401	< 2.2e-16	***
linear:sizeLay	1	0.0284	0.0284	34.5312	1.57E-08	***
learnF:linear:sizeLayer	5	0.1503	0.0301	36.5029	< 2.2e-16	***
Residuals	216	0.1778	0.0008			

Table 15. ANOVA of the factorial experiment

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

4.6. Artificial Neural Networks: python implementation

4.6.1. Why Python?

Although R is a very powerful tool to train nets and analyse its performance, it doesn't deliver a user-friendly interface for the end user. Moreover, some of the main neural network packages – such as Theano[1], Pylearn[2], Caffe[3] etc. – do not offer support for R. Thus, to get over those problems, the programming language had to be switched into a more versatile option. Two main languages offered a solid alternative to R: C++ and Python. Although C++ is largely used, implementing a neural network proof to be a tough task, whereas in Python codes are simpler and clearer. Moreover, Python also allows writing simple user interfaces and the pipeline defined could be easily redirected if needed.

When coming to data, Python also have a clear advantage over R as it allows parsing a file within a few lines. In addition, all the pre-process of the data has been done in Python, so it was the natural evolution of the workflow.

In addition to the language, the SDK to develop the application is a key element. In this case, all the Python codes and graphics have been run under the iPython Notebook[4]. This tool has the ability to run the code in independent blocks, always maintaining the value of the variables in a *debug* mode.

[1] http://deeplearning.net/software/theano/
[2] https://github.com/lisa-lab/pylearn2/
[3] http://caffe.berkeleyvision.org/
[4] http://ipython.org/notebook.html

4.6.2. Steps before training

To train the neural network, in any of the studied cases, it is necessary to load all the data into a python format. To do so, two functions have been implemented to parse the data from a *csv* format into a two-dimensional *NumPy*[5] array and normalize the data.

The first one, reads a given file, and returns a *NumPy* array with all the information. By means of the *normalization* flag, the user can specify if the data should be normalized. By default this option is set as *True* and is computed in an independent function. The separator to parse the data can also be set, although the default character is ";".

```
def readFile(fileIn, isHeader = True, normalization = True, separator = ";"):
    count = 0
    dataLog = []
    for line in fileIn:
        if isHeader:
            isHeader = False
        else:
            try:
                line = line.split("\r\n")[0]
                line = line.split(separator)
                row = []
                for num in line:
                    row.append(float(num))
                dataLog.append(np.array(row))
            except:
                print "Error parsing line [%d]: %s" % (count, line)
            count += 1
    npDataLog = np.array(dataLog)
    if normalization:
        npDataLog = normData(npDataLog)
    return npDataLog
```

[5] http://www.numpy.org/

Scales the data by columns into a range between 0 and 1. To normalize the data the minimum is subtracted and all the values are divided by the maximum. The values used to implement this normalization are stored in a file (by default normVal.csv), in order to undo the normalization once the prediction is done.

```
def normData(argray, normPath = "normVal.csv"):
    vNorm = []
    size = array.shape
    normFile = open(normPath, 'w')
    for idx in range(size[-1]):
        normFile.write(str(array[:,idx].min()) + " " + str(array[:,idx].max() +
"\n")
        array[:,idx] -= array[:,idx].min()
        array[:,idx] /= array[:,idx].max()
    normFile.close()
    return array
```

4.6.3. PyBrain

In a first approach, the neural network training has been implemented using a well-know library: *PyBrain*[6]. Although there are alternative libraries in Python for this purpose, PyBrain is one of the most complete and has a complete documentation with many examples. Moreover, it is easy to create a multilayer perceptron with very few lines and many of its training parameters can be set. The only inconvenience of *PyBrain* is that it requires a specific type of data structure and so, a function to switch from the raw information to the specific format. This class is called SupervisedDataSet and requires two parameters to be initialized: the number of the inputs and outputs.

```
def createDataSet(array):
    ds = SupervisedDataSet( lenInput, lenOutput )
    for row in array:
        ds.appendLinked(row[:-1], row[-1])
    return ds
```

4.6.3.1. PyBrain code

When working with PyBrain, it necessary to define three key elements:

- The layers (morphology of the different layers)
- The structure of the net (number of neurons and their connections)
- The trainer (algorithm to minimize the weights)

[6] http://pybrain.org/

In the code below, those three elements are imported from the PyBrain package, the data is loaded from a *csv* file, the sizes of the different layers of the net are specified and the sets are created in a certain proportion.

```
from pybrain.structure import SigmoidLayer, LinearLayer
from pybrain.tools.shortcuts import buildNetwork
from pybrain.supervised.trainers import BackpropTrainer


dataFile = open('/Users/sergi/Desktop/data_resp.csv', 'r')
data = readFile(dataFile, separator = ";")

lenInput = len(data[0]) - 1
hiddenUnits = 25
lenOutput = 1

ds = createDataSet(data)
te, tr = ds.splitWithProportion( 0.15 )
```

The net is defined, with 25 hidden units and a sigmoid function as activation. The back propagation trainer is initialized, with the flag verbose to evaluate the intermediate states of the training.

```
net = buildNetwork(lenInput, hiddenUnits, lenOutput, bias = True, hiddenclass =
SigmoidLayer, outclass = LinearLayer)

trainer = BackpropTrainer(net, tr, verbose = True)
```

To train the net, a fixed number of epochs is fixed. To track the training, the function prompts the error of the train and the test, for each epoch.

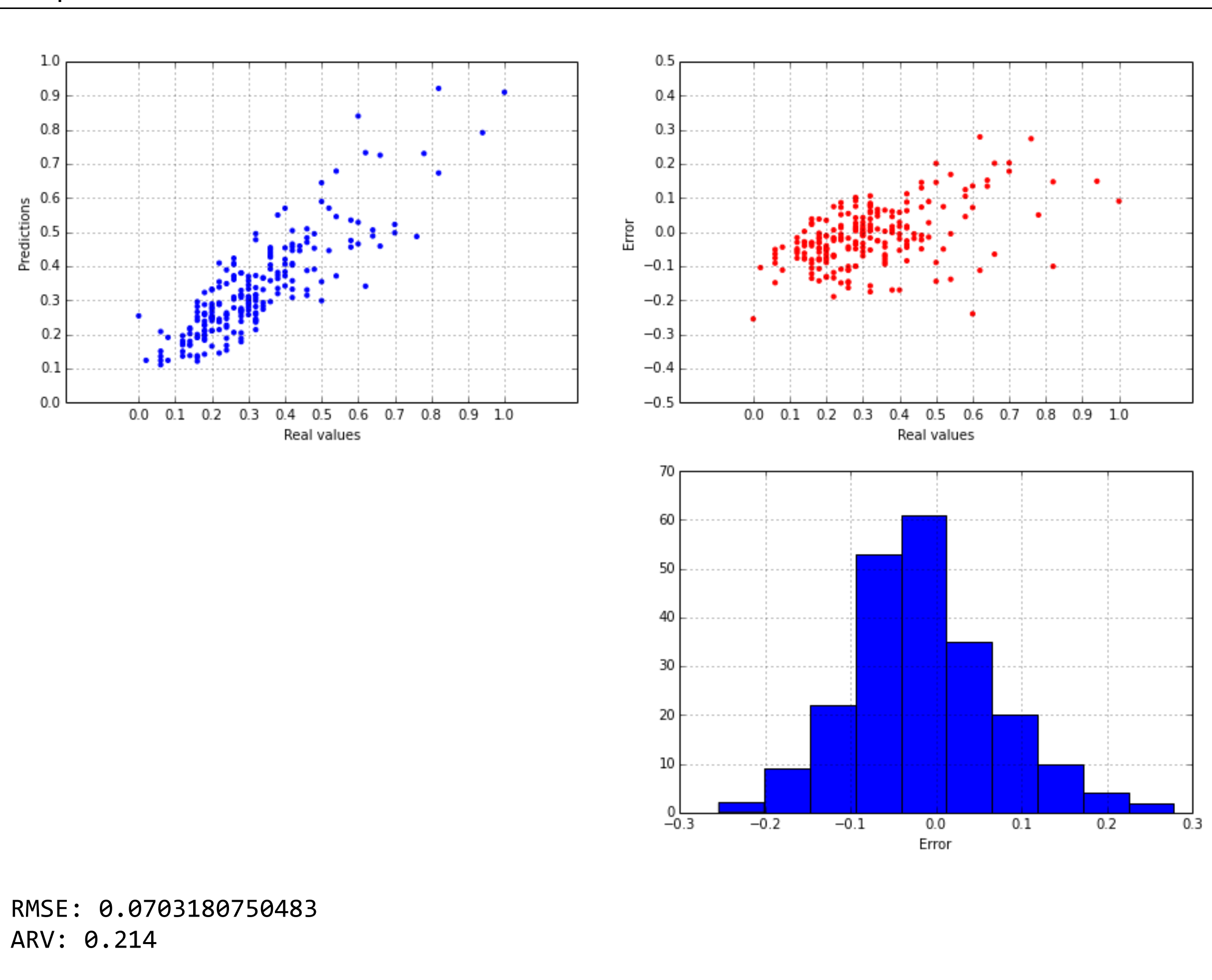
```
trainError = []
testError = []
for iteration in range(200):
    error = trainer.train()
    trainError.append(error)

    listError = []
    predictions = []
    for value in te['input']:
        predictions.append(net.activate(value))
    for j in range(len(predictions)):
        listError.append((predictions[j] - te['target'][j]))
    listError = np.array(listError)
    rmse = float(np.sqrt(np.sum(listError**2)/len(listError)))
    testError.append(rmse)
```

To understand completely the quality of the training, two graphics are plotted. The first one shows the predictions versus the real values. The second one shows the error of the prediction against the real value. In addition to the graphics, the RMSE and ARV are calculated.

```
import pylab
import numpy as np

predictions = []
for value in te['input']:
    predictions.append(net.activate(value))

# Plot the prediction against the real value
axisMin = 0.0
axisMax = 1.0
pylab.rcParams['figure.figsize'] = (15, 10)
pylab.subplot(2,2,1)
pylab.scatter(te['target'], predictions, s=10, color = 'blue', label = 'NN
output')
pylab.xticks(pylab.linspace(axisMin, axisMax, 11, endpoint=True))
pylab.yticks(pylab.linspace(axisMin, axisMax, 11, endpoint=True))
pylab.xlabel("Real values")
pylab.ylabel("Predictions")
pylab.grid()

# Plot the error against the real value
pylab.subplot(2,2,2)
pylab.scatter(te['target'], te['target'] - predictions, s=10, color = 'red', label
= 'NN output')
pylab.xticks(pylab.linspace(axisMin, axisMax, 11, endpoint=True))
pylab.yticks(pylab.linspace(-axisMax/2, axisMax/2, 11, endpoint=True))
pylab.xlabel("Real values")
pylab.ylabel("Error")
pylab.grid()

# Plot an histogram of the error.
pylab.subplot(2,2,3)
pylab.hist(te['target'] - predictions)
pylab.xlabel("Error")
pylab.grid()
pylab.show()

# Compute the RMSE
listError = []
for j in range(len(predictions)):
    listError.append((predictions[j] - te['target'][j]))
error = np.array(listError)
rmse = float(np.sqrt(np.sum(error**2)/len(error)))
print "RMSE:", rmse
```

```
# Compute the ARV
teMean = 0
for i in range(len(te)):
    teMean += te['target'][i][0]
teMean /= len(te)
arv = np.sum(error**2)/np.sum((te['target'] - teMean)**2)
print "ARV: %.3f" % arv
```

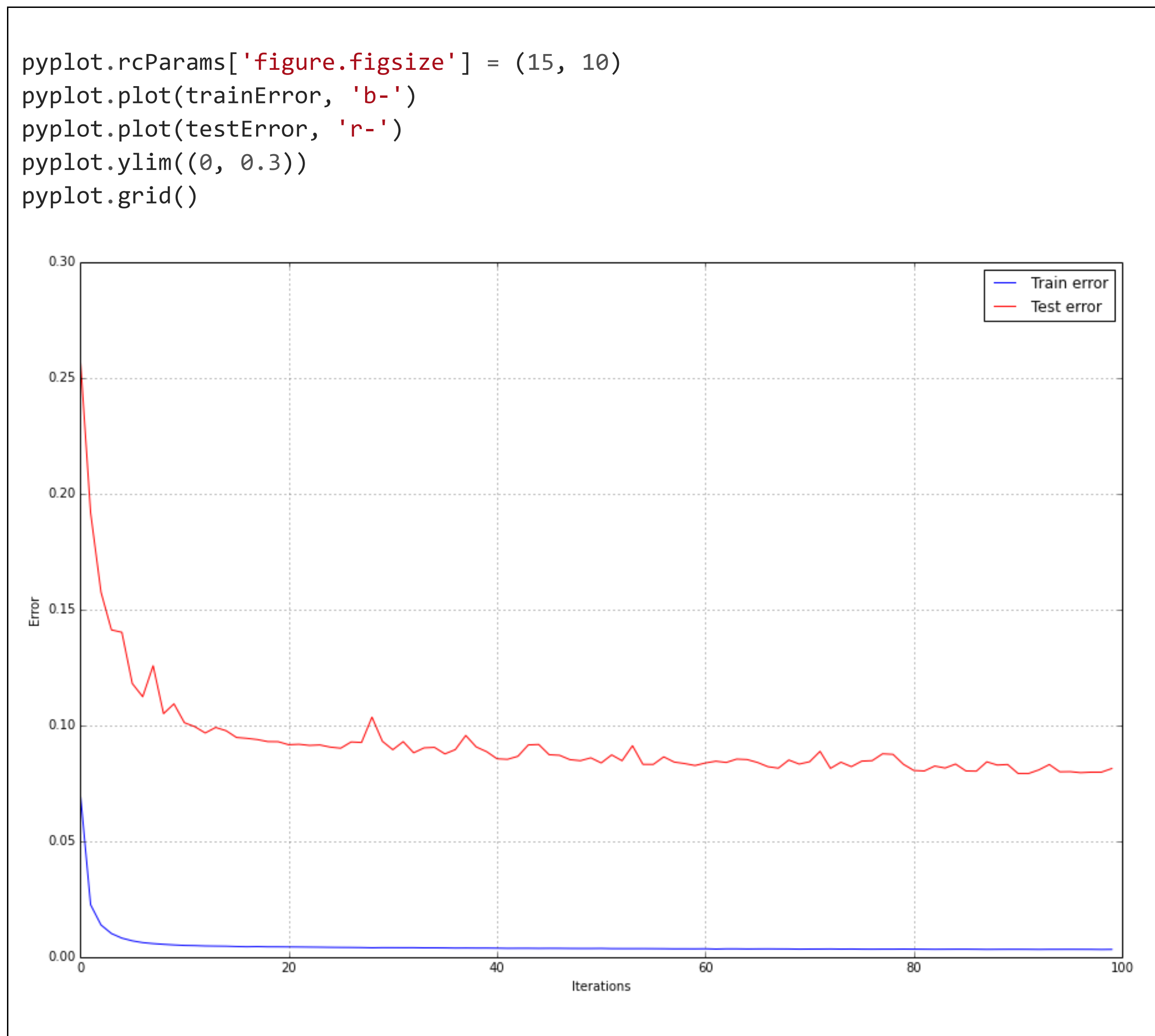
Output.



Plot 40. Training error with the PyBrain package.

As evaluated before those values are typical results for this configuration. The errors also confirm the assumption that huge values are more unlikely to be predicted.

Finally, the errors obtained during the training are plotted. It is clear from the graphic that the training has achieved a local (either minimum or absolute) and the accuracy doesn't improve with the number of epochs.



Plot 44. Train and test error with the PyBrain package.

4.6.4. Class implementation

Although the *PyBrain* library is a powerful tool to work with, some elements of the network structure and its behaviour remain uncontrolled by the user. Thus, writing a Python class over *NumPy*, to define and train the network, is the best option to understand the whole process and control all the variables.

4.6.4.1. ANN code

Import libraries to process the data and draw the graphics. Nearly all the computational calculus have to be done by means of the NumPy package, however some functions such as the hyperbolic tangent or randomizing the data are implemented in other Python packages.

```
import math
import pylab
import random
import numpy as np
```

Functions to define the activation functions (sigmoid and hyperbolic tangent) and their derivatives. Those functions are required to process the entire algorithm when predicting and computing the back propagation.

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def dsigmoid(y):
    return y * (1.0 - y)

def tanh(x):
    return math.tanh(x)

def dtanh(y):
    return 1 - y*y
```

Once the librairies are imported and the activation functions are defined, the Multilayer Perceptron Class can be defined with its attributes and methods. The basic structure of the this class is defined below:

Attributes to define the network:

- Solver parameters (iterations, learning rate, momentum and rate decay)
- Size of the layers
- Weight values
- Temporal variables to compute the back propagation.

Methods to use the network:

- Feed forward computation
- Back propagation
- Test
- Train
- Predict

```
class MLP_NeuralNetwork(object):
        # initialize parameters
        self.iterations = iterations
        self.learning_rate = learning_rate
        self.momentum = momentum
        self.rate_decay = rate_decay

        # initialize arrays
        self.input = input + 1 # add 1 for bias node
        self.hidden = hidden
        self.output = output

        # set up array of 1s for activations
        self.ai = [1.0] * self.input
        self.ah = [1.0] * self.hidden
        self.ao = [1.0] * self.output

        # create randomized weights
        # use scheme from 'efficient backprop to initialize weights
        input_range = 1.0 / self.input ** (1/2)
        output_range = 1.0 / self.hidden ** (1/2)
        self.wi = np.random.normal(loc = 0, scale = input_range, size =
(self.input, self.hidden))
        self.wo = np.random.normal(loc = 0, scale = output_range, size =
(self.hidden, self.output))

        # create arrays of 0 for changes
        # this is essentially an array of temporary values that gets updated at
        # each iteration
        # based on how much the weights need to change in the following iteration
        self.ci = np.zeros((self.input, self.hidden))
```

```
        self.co = np.zeros((self.hidden, self.output))

    def feedForward(self, inputs):
        if len(inputs) != self.input-1:
            raise ValueError('Wrong number of inputs')

        # input activations
        for i in range(self.input -1): # -1 is to avoid the bias
            self.ai[i] = inputs[i]

        # hidden activations
        for j in range(self.hidden):
            sum = 0.0
            for i in range(self.input):
                sum += self.ai[i] * self.wi[i][j]
            self.ah[j] = tanh(sum)

        # output activations
        for k in range(self.output):
            sum = 0.0
            for j in range(self.hidden):
                sum += self.ah[j] * self.wo[j][k]
            self.ao[k] = sigmoid(sum)

        return self.ao[:]

    def backPropagate(self, targets):
        if len(targets) != self.output:
            raise ValueError('Wrong number of targets!')

        # calculate error terms for output
        # the delta tell you which direction to change the weights
        output_deltas = [0.0] * self.output
        for k in range(self.output):
            error = -(targets[k] - self.ao[k])
            output_deltas[k] = dsigmoid(self.ao[k]) * error

        # calculate error terms for hidden
        # delta tells you which direction to change the weights
        hidden_deltas = [0.0] * self.hidden
        for j in range(self.hidden):
            error = 0.0
            for k in range(self.output):
                error += output_deltas[k] * self.wo[j][k]
            hidden_deltas[j] = dtanh(self.ah[j]) * error

        # update the weights connecting hidden to output
        for j in range(self.hidden):
            for k in range(self.output):
                change = output_deltas[k] * self.ah[j]
```

```
                self.wo[j][k] -= self.learning_rate * change + self.co[j][k] *
self.momentum
                self.co[j][k] = change

        # update the weights connecting input to hidden
        for i in range(self.input):
            for j in range(self.hidden):
                change = hidden_deltas[j] * self.ai[i]
                self.wi[i][j] -= self.learning_rate * change + self.ci[i][j] *
self.momentum
                self.ci[i][j] = change

        # calculate error
        error = 0.0
        for k in range(len(targets)):
            error += 0.5 * (targets[k] - self.ao[k]) ** 2
        return error

    def test(self, patterns):
        for p in patterns:
            print(p[1], '->', self.feedForward(p[0]))

    def train(self, patterns):
        # N: learning rate
        for i in range(self.iterations):
            error = 0.0
            random.shuffle(patterns)
            for p in patterns:
                inputs = p[0]
                targets = p[1]
                self.feedForward(inputs)
                error += self.backPropagate(targets)
            with open('error.txt', 'a') as errorfile:
                errorfile.write(str(error) + '\n')
                errorfile.close()
            if i % 10 == 0:
                print('error %-.5f' % error)
            # learning rate decay
            self.learning_rate = self.learning_rate * (self.learning_rate /
(self.learning_rate + (self.learning_rate * self.rate_decay)))

    def predict(self, X):
        predictions = []
        for p in X:
            predictions.append(self.feedForward(p))
        return prediction
```

Function to create the sets to train and test the network. The data as specified by the flag *testProportion*. This variable accepts values between 0 and 1.

```
def createSets(dataLog, testProportion):
    """
    Given a set and a test proportion, returns 4 sets (trX, trY, teX, teY).
    """
    setSize = len(dataLog)
    testSize = int(setSize*testProportion)
    trX = dataLog[:-testSize,:-1]
    trY = dataLog[:-testSize,-1].reshape((setSize - testSize, 1))
    teX = dataLog[-testSize:,:-1]
    teY = dataLog[-testSize:,-1].reshape((testSize, 1))
    return trX, trY, teX, teY
```

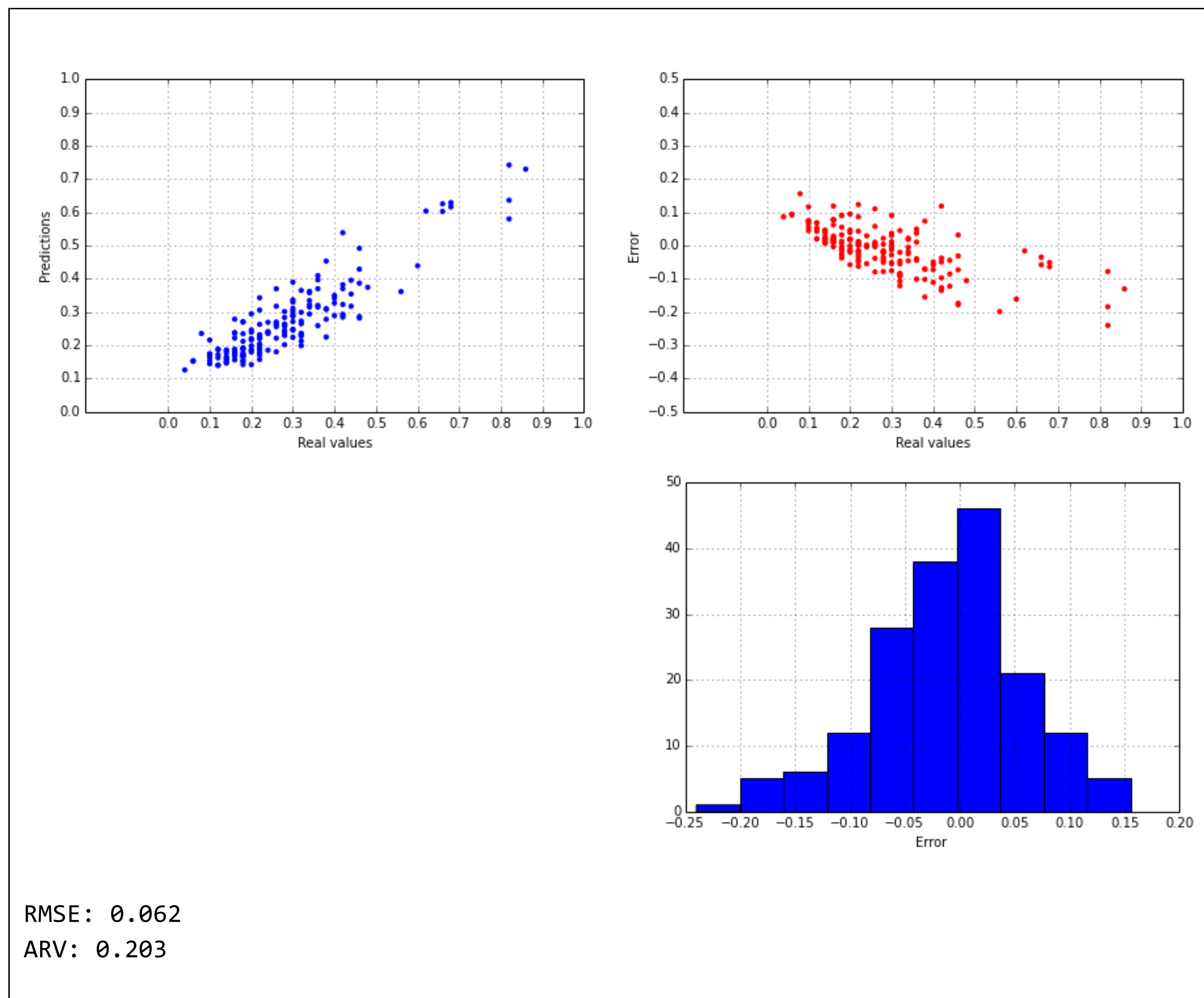
Main code. Read a file with the data to train, parse this data, create the required sets, define a neural network and train the network.

```
dataFile = open('data_resp.csv', 'r')
data = readFile(dataFile, separator = ";")
trX, trY, teX, teY = createSets(data, 0.15)
NN = MLP_NeuralNetwork(data.shape[1] - 1, 20, 1, iterations = 200, learning_rate =
0.6, momentum = 0.8, rate_decay = 0.01)
NN.train(zip(trX, trY))
```

Plot the predictions against the real values, and the errors against the real values. Compute the RMSE and the ARV.

```
# Plot the prediction against the real value
teY = teY.reshape(teY.shape[0])
axisMin = 0.0
axisMax = 1.0
pylab.rcParams['figure.figsize'] = (15, 10)
pylab.subplot(2,2,1)
pylab.scatter(teY, pred, s=10, color = 'blue', label = 'NN output')
pylab.xticks(pylab.linspace(axisMin, axisMax, 11, endpoint=True))
pylab.yticks(pylab.linspace(axisMin, axisMax, 11, endpoint=True))
pylab.xlabel("Real values")
pylab.ylabel("Predictions")
pylab.grid()

# Plot the error against the real value
pylab.subplot(2,2,2)
pylab.scatter(teY, pred - teY, s=10, color = 'red', label = 'NN output')
pylab.xticks(pylab.linspace(axisMin, axisMax, 11, endpoint=True))
pylab.yticks(pylab.linspace(-axisMax/2, axisMax/2, 11, endpoint=True))
pylab.xlabel("Real values")
pylab.ylabel("Error")
pylab.grid()
pylab.show()

# Plot an histogram of the error.
pylab.subplot(2,2,4)
pylab.hist(pred - teY)
pylab.xlabel("Error")
pylab.grid()
pylab.show()
```

Output.




```
RMSE: 0.062
ARV: 0.203
```

Plot 45. Training error with the MLP class.

In addition to the methods described for the class, there are some implementations that are required in order to manage the neural network in futures developments. Essentially, those methods are saving and loading a neural network, and should be added to the class.

```
Def storeNN(self, pathFile):
    np.savez(pathFile, self.ai, self.ah, self.wi, self.wo)

def loadNN(self, pathFile):
   npzfile = np.load(pathFile)
    arrayList = npzfile.files
    arrayList.sort()
    self.ai = npzfile[arrayList[3]]
    self.ah = npzfile[arrayList[4]]
    self.wi = npzfile[arrayList[5]]
    self.wo = npzfile[arrayList[6]]
```

4.6.4.2. Python conclusions

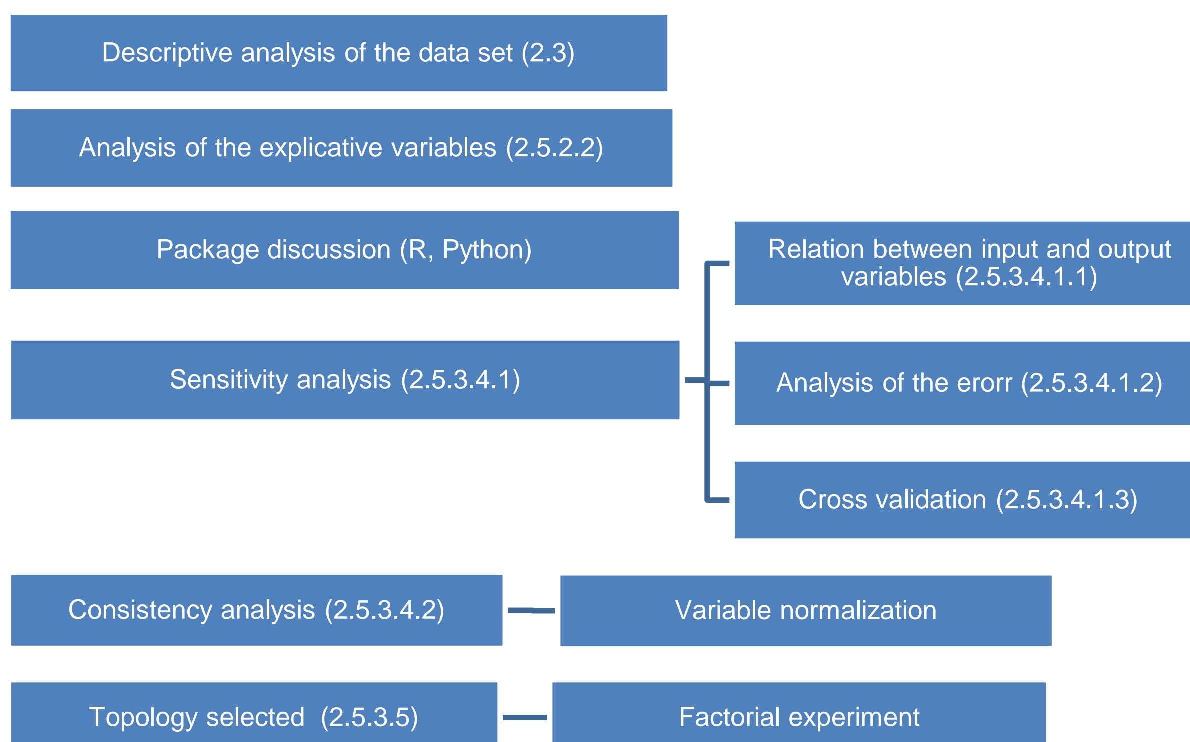
After analysing both methods to train a neural network and predict accurate outputs, it becomes clear that Python is a very powerful tool to develop this task. PyBrain has proved to be a solid option, with many features and a complete set of algorithm and morphologies. However, the usage of a class, that implements a Multilayer Perceptron, also gives a bigger control over the process, which is necessary for the development of a user interface. Simple operations such as adding information to the database, or storing the values for the normalization of the different parameters, can be easily done defining a suitable standard. In terms of quality of the result, Python is as strong as R both in computational speed and accuracy.

All those reasons proof that a language such as Python, which is not specifically thought for mathematical computation, can be adapted to behave as a perfect package for this task. Moreover, the flexibility of Python allows the partial usage of the pipeline (e.g. reading the database, normalizing the data, plotting the errors etc.) even if the algorithm is switched to a Support Vector Regression or any other machine-learning paradigm.

5. Conclusions

Pursuing the best model to predict the emergencies, occurring during the year, in the Hospital Universitari Dr. Josep Trueta de Girona, several methodologies and algorithms have been applied: Starting with a temporal series analysis which was not able to accurately capture the flow of the process; followed with a deep study of the artificial neural networks applied to a regression problem. Using this algorithm, contained in the machine learning analysis, has bring two important work paths: develop a based and reproducible methodology and adapt it to solve the presented problem.

There are several papers using artificial neural networks for classification purposes and even for linear regressions, but there are just a few regarding nonlinear predictions for temporal series like the one presented in this project. The lack of a well-defined methodology to face this kind of process has lead us to try to define, as detailed as possible, a consistent work frame for future projects pursuing similar objectives. Therefore, and according to the discussion described in the core of the project, the following steps have been defined:



ETSEIB

Despite the model is capable of capture the main flow of the process, it is failing in the forecast of the outliers which would have represent an important improvement compared with a regression method. Even though different methodologies and explicative variables have been applied to correct this issue, they have not offered good solutions. The algorithms that presented a lower error levels due the capture of some of this outliers (as using the *Rprop* learning function) in the training, have failed in the test due to the over fitting made. As a last possible solution, knowing the seasonality of the process, two neural networks have been built (each one according to the frequency of emergencies: low for spring and summer or high for autumn and winter) and trained according this definition. The results are not shown as they are worse than the ones using the previous methodology due to by partitioning the problem, there are too few registers to train the algorithm.

6. Economic study

In this section, the costs and profits of the project are discussed. First, a brief impact analysis regarding the implementation of the project into the Hospital Universitari de Girona Dr. Josep Trueta is performed. Secondly all the associated costs of the development of the project are accurate listed.

Assuming that the algorithm offers good forecasts, the hospital management could you use it in order to schedule and scale both resource and personnel actives of the hospital. The application provided to the management team would help them know a good approximation of the emergencies left to come at the emergency department once the information from the morning (first strip) has been introduced to the algorithm via the plain application. It is rough to calculate the exact benefits (in cash) for the hospital even though it is easy to identify it would suppose a great help for the emergency department logistics once a good estimation of the volume of emergencies has been predicted.

The project's costs can be disaggregated due several concepts:

- Human resources
  - Consulting hours to analyse, onside, the problem to solve and to investigate its related covariates.
  - Salary associated to the development of the algorithm according an engineering salary.
  - Insurances.
- Energy consumption: electricity (charge the laptop and light for the office).
- Infrastructure
  - Hardware: laptop, screen, mouse and keyboard.
  - Software: licenses for using the programs.
  - Location.

There are several costs than can be discounted as they have a low impact in the total budget of the project. In this expenses category the following concepts are included:

- Infrastructure – location: due all the project has been developed in the engineer's own place, and occasionally in public areas (such as public libraries or the university), the costs associated with the location and with the energy consumption have not been taken in account for the total cost calculation. The hardware costs are also not used for the costs calculations as the development

of the project does not imply a depreciation of this hardware, which is used for private reasons as well.

- All the insurance costs have been neglected as well. Knowing that there already is a student insurances paid.

The costs of all other concepts have been taken into account. The most representative ones are the ones related to engineering hours. This concept has been partitioned in two different types of costs. From one side, the consulting spent hours spent on side, in the hospital, in order to inquire and gather as much information as possible about the future modeled process. Once this hours satisfies the requirements gathering, the development hours associated to engineering costs have to be counted.

The total amount of hours dedicated to develop this project are the ones corresponding to the hours scheduled to achieve the equivalence with 12 ECTS's credits. In this case it implies among 360 hours which mean, approximately, nine weeks working eight hours per day.

Another cornerstone to calculate the costs in the project is the price of the licenses for the software used during the development. In this case R and its framework extension, Rstudio, have been used for the analysis and the development of the algorithm. Then, in order to be able to tune as much as possible the algorithm and also to get it ready for the applicative, Python has been used. The cost of this software is null as all of them belongs to free software which has not to be paid for the development of the project.

Another licenses used have been the ones corresponding to Windows and Microsoft Office which are included in the price of buying the laptop.

Salary concepts	Cost / hour (€/h)	Total time (h)	Total cost (€)
Consulting	25	40	1000
Development	20	320	6400
Total	-	-	9600

Table16. Salary costs

Hardware	Total cost (€)	Amortization time (months)	Duration of use (months)	Cost associated (€)
Laptop	2200	84	2	50
Others (mouse, keyboard…)	150	42	2	8
Total	-	-	-	58

Table 17. Infrastructure costs

Project total cost is equal to 9600 + 58 = 9658 €.

7. Environmental study

The environmental impact of the system developed by the current project is not significant. It is a fact that, by taking into account the decrease of the hospital resources spend due a better prediction of the volume of emergencies, would bring a less negative environmental effect. The real values of the reductions in environmental costs that this system could mean cannot be accurately estimated in the scope of this project.

Another keystone that has to be analysed is the environmental cost of the project itself. Even though there is an undoubtedly energetic cost due the utilization of electronic equipment (in this case a laptop), it can be neglected, as the cost of the energy consumption of the laptop was not taken into account in the economic study.

8. Bibliography

Montserrat Pepió. Seres temporales. 2ª ed. Barcelona: Edicions UPC, 2002.

Mª Luisa Pérez, Quintín Martín. Aplicaciones de las redes neuronales artificiales a la estadística. Madrid: Hespérides, S.L, 2003.

Mª Pilar González. Análisi de series temporales: Modelos ARIMA. País Vasco: Facultad de Ciencias Económicas y Empresariales, 2009.

J.P. Marques de Sá. Pattern Recognition. Oporto:Springer, 2001.

Department of Computer Science Mangalore University. “PERFORMANCE ANALYSIS OF NEURAL NETWORK MODELS FOR OXAZOLINES AND OXAZOLES DERIVATIVES DESCRIPTOR DATASET”. Vol.3 No.6, November 2013.

Christoph Bergmeir, José M. Benítez. “Neural Networks in R Using the Stuttgart Neural

Network Simulator: RSNNS”. Vol. 46, Issue 7, January 2012.

Jonathan D.Cryer, Kung-Sik Chan. Time Series Analysis, 2nd ed. USA: Springer, 2008.

Sarah Thompson. R is my friend. [Checked: July 2014]. http://beckmw.wordpress.com/2013/08/12/variable-importance-in-neural-networks/

Quick-R (accessing the power of R). [Checked: July 2014, August 2014]. http://www.statmethods.net/advstats/timeseries.html

Avril Chohlan. Little Book of R for Time Series. Cambridge, U.K. [Checked: May 2014, July 2014]. Disponible a: http://a-little-book-of-r-for-time-series.readthedocs.org/en/latest/

Cross Validated. [Checked: July 2014, August 2014]. http://stats.stackexchange.com/

Andreas Zell, Günter Mamier, Michael Vogt. Stuttgart Neural Network Simulator User Manual, Version 4.1. [Checked: February 2015]. http://www.ra.cs.uni-tuebingen.de/SNNS/UserManual/UserManual.html

Wikipedia. [Checked January 2015]. http://en.wikipedia.org/wiki/Cross-validation_(statistics)

Robert Griñó Cubero. Neural Networks for water demand time series forecasting. Institut de Cibernètica (CSIC-UPC).

Theophano Mitsa. (2010). Temporal Data Mining. USA. Chapman & Hall book.

Hecht-Nielsen, R. (June 22, 1989). “Theory of the backpropagation neural network”. 593 – 605

vol.1.

A.T.C. Goh. (1995). “Back-propagation neural networks for modeling complex systems”. Volume 9, Issue 3, 1995, Pages 143–151.

Alex J. Smola, Bernhard Schölkopf. (1998). [Checked: January 2015] “A Tutorial on Support Vector Regression”. http://www.svms.org/regression/SmSc98.pdf

Scikit-learn. Support Vector Regression. [Checked: February 2015]. http://scikit-learn.org/stable/auto_examples/svm/plot_svm_regression.html

iPython. [Checked: October 2014]. “The Ipython Notebook”. http://ipython.org/notebook.html

Numpy manual. [Checked: November 2014]. http://docs.scipy.org/doc/numpy/reference/

PyPlot Summary. [Checked: November 2014]. http://matplotlib.org/api/pyplot_summary.html

PyBrain docs. [Checked: October 2014]. http://pybrain.org/docs/

9. ANNEX 1

9.1. Core of the algorithm

Once the different parts of the code that integrates the neural network algorithm have been explained, is time to take a look into the core of this code in order to know how it is internally organized.

#Load the data

```
dataTot<-read.csv2(file="respiratory.csv", header=T, sep=";",dec=".")

idresp=9

data=dataTot[,1:idresp]
```

#Normalize the data

```
data[,1]=data[,1]/max(data[,1])

data[,2:5]=data[,2:5]/max(data[,2:5])

data[,6]=data[,6]/max(data[,6])

data[,idresp]=data[,idresp]/max(data[,idresp])
```

#Initialization of the library and data set

```
library(RSNNS)

set.seed(12345678)

inputs<-data[,1:(ncol(data)-1)]

outputs<-data[,ncol(data)]

train_old<-data.frame(cbind(inputs,outputs))

train=train_old
```

#Definitions (train and validation sets, training blocks, ARV, MSE and error)

```
train_set<-train[1:1273,]

validation_set<-train[1274:1454,]

block<-7

fac=1

nveg=5

TARV=matrix(0,ncol=nveg,nrow=block)

TMSE=matrix(0,ncol=nveg,nrow=block)

error=matrix(0,ncol=nveg,nrow=nrow(train_set))



ARV=function(a,p) sum((a-p)^2)/sum((a-mean(a))^2)

MSE=function(a,p) sqrt(mean((a-p)^2))
```

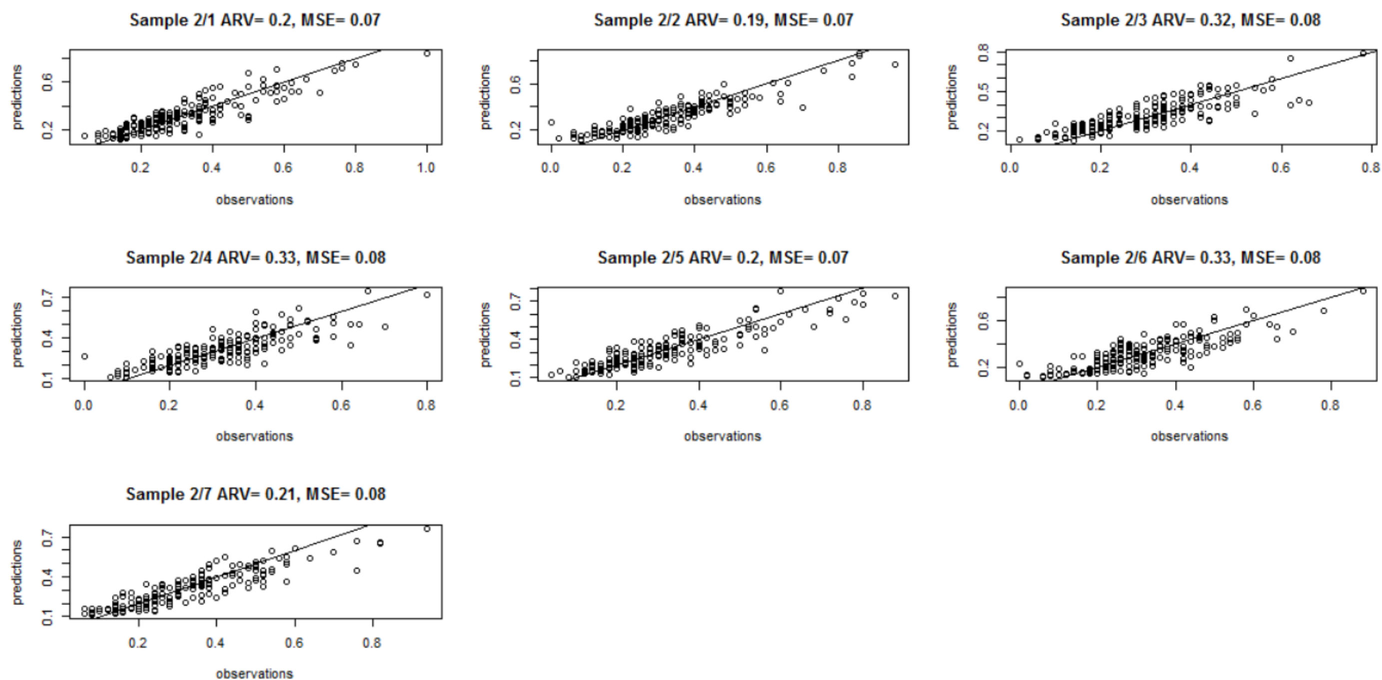
Loop 1: setting the parameter *nveg,* the number of times that the multi-layer perceptron (mlp) is going to compute the training (with different initialization parameters) is fixed. Inside this first loop there is the *sample* function which randomly organizes the data from the training set (keeping out the one for the validation) into the defined number of blocks, *block*, that are going to be used in the Loop 2.

The parameter *block*, indicates in how many blocks should the training set be partitioned for performing the cross validation using the k-folders non exhaustive technique.

The calculations for each block are done in the second loop, # Loop 2, where the values for ARV and MSE are registered in the predefined matrices for each iteration.

For each one of the *nveg* (five in this example) iterations, the neural network trains *block* times with different data (according to the k-fold technique) the algorithm (in this case, seven). In the example below, there are the seven different results of the training for the *nveg* equals to two.

This methodology is applied in order to prove the consistency of the algorithm which, according to numerous tests, it is consistent with the data that is being used.



Plot 46. Predictions vs Observations

```
for (ii in 1:nveg){

  samp<-sample(1:block, size=nrow(train_set), replace=T, prob=rep(1/block, block))



  par(mfrow=c(3,3))

  for (i in (1:block)){

    id<-which(samp==i)

    dataTrain<-train_set[-id,]

    dataTest<-train_set[id,]

    n_in<-names(train_set)[1:(ncol(data)-1)]

    n_out<-names(train_set)[ncol(data)]
```

```
nn<-mlp(x=dataTrain[,n_in],      y=dataTrain[,n_out],      size=10,      maxit=1000,      linOut=F,
learnFunc=BackpropWeightDecay)

    predictions<-fac*predict(nn, data.frame(dataTest[,n_in]))

    observations<-fac*dataTest$outputs



    (TARV[i,ii]<-ARV(observations,predictions))

    (TMSE[i,ii]<-MSE(observations,predictions))

    plot(observations,predictions,

         main=paste("Sample      ",ii,"/",i,"      ARV=      ",round(TARV[i,ii],2),",      MSE=
",round(TMSE[i,ii],2),sep=""))

    abline(c(0,1))

    print(summary(mod<-lm(observations~predictions)))

    error[id,ii]<-resid(mod)

  }

  par(mfrow=c(1,1))

}
```

#Matrix of the highest errors

```
pes=apply(error,1,function(el) sum(el^2))

dataTot[order(pes,decreasing=T)[1:20],]

error[order(pes,decreasing=T)[1:20],]
```

Neural network training for the validation

```
dataTrain<-train_set

dataTest<-validation_set

n_in<-names(train_set)[1:(ncol(data)-1)]

n_out<-names(train_set)[ncol(data)]



nn<-mlp(x=dataTrain[,n_in],  y=dataTrain[,n_out],  size=25,  maxit=1000,  linOut=F,
learnFunc='BackpropWeightDecay')
```

Prediction for the validation set

```
predictions<-fac*predict(nn, data.frame(dataTest[,n_in]))
```

Results and error calculation

```
observations<-fac*dataTest$outputs

(TARV1<-ARV(observations,predictions))

(TMSE1<-MSE(observations,predictions))

plot(observations,predictions,

     main=paste("Validation ARV= ",round(TARV1,2),", MSE= ",round(TMSE1,2),sep=""))

abline(c(0,1))

summary(mod<-lm(observations~predictions))
```

```
#Starts at 8th of January 2010

#Data load

dataTot<-read.csv2(file="reesp_2.csv", header=T, sep=";",dec=".")

idresp=9

nveg=3

data=dataTot[,1:idresp]


data[,1]=data[,1]/max(data[,1])

data[,2:5]=data[,2:5]/max(data[,2:5])

data[,6]=data[,6]/max(data[,6])

data[,idresp]=data[,idresp]/max(data[,idresp])
```

```
library(RSNNS)

set.seed(12345678)

inputs<-data[,1:(ncol(data)-1)]

outputs<-data[,ncol(data)]

train=data.frame(cbind(inputs,outputs))


fi=365*3-7+1

train_set<-train[1:fi,]

validation_set<-train[(fi+1):1454,]


TARV=matrix(0,ncol=nveg,nrow=block)

TMSE=matrix(0,ncol=nveg,nrow=block)

error=matrix(0,ncol=nveg,nrow=nrow(train_set))


ARV=function(a,p) sum((a-p)^2)/sum((a-mean(a))^2)

MSE=function(a,p) sqrt(mean((a-p)^2))
```

9.2. Factorial experiment code

```
dataTrain<-train_set

dataTest<-validation_set

n_in<-names(train_set)[1:(ncol(data)-1)]

n_out<-names(train_set)[ncol(data)]



fac=1

rep=3

k=0

ntot=6*2*3*rep

res=data.frame(learnF=rep(NA,ntot),linear=rep(NA,ntot),sizeLay=rep(0,ntot),ii=rep(0,ntot),TAR
V1=rep(0,ntot),TMSE1=rep(0,ntot),TARV2=rep(0,ntot),TMSE2=rep(0,ntot))
```

```
For                                         (learnF                                       in
c("Std_Backpropagation","BackpropBatch","BackpropChunk","BackpropMomentum","BackpropWeightDec
ay","Rprop")){

  for (linear in c(TRUE,FALSE)){

    for (sizeLay in c(2, 10, 25, 50, 100)){

      k=k+1

      for (ii in 1:rep){

          nn<-mlp(x=dataTrain[,n_in],    y=dataTrain[,n_out],    size=sizeLay,    maxit=1000,
linOut=linear, learnFunc=learnF)

          predictions<-fac*predict(nn, data.frame(dataTrain[,n_in]))

          observations<-fac*dataTrain$outputs

          TARV1<-ARV(observations,predictions)

          TMSE1<-MSE(observations,predictions) ...
```

```
… plot(observations,predictions,
              main=paste("Sample      ",ii,"/",k,"      ARV=      ",round(TARV1,2),",      MSE=
",round(TMSE1,2),sep=""))
          abline(c(0,1))
          print(summary(mod<-lm(observations~predictions)))
          predictions<-fac*predict(nn, data.frame(dataTest[,n_in]))
          observations<-fac*dataTest$outputs
          TARV2<-ARV(observations,predictions)
          TMSE2<-MSE(observations,predictions)
          plot(observations,predictions,
               main=paste("Sample      ",ii,"/",k,"      ARV=      ",round(TARV2,2),",      MSE=
",round(TMSE2,2),sep=""))
          abline(c(0,1))
          print(summary(mod<-lm(observations~predictions)))
          #error[id,ii]<-resid(mod)
          res[(k-1)*rep+ii,]=c(learnF,linear,sizeLay,ii,TARV1,TMSE1,TARV2,TMSE2)
        }
      }
   }
}
```

```
write.csv2(res,"resultats.csv")
```

9.3.

9.4. PyBrain documentation

Functions implemented in PyBrain used to develop the software for this project.

9.4.1. Create and process the data sets

class pybrain.datasets.supervised.**SupervisedDataSet**(*inp*, *target*)

SupervisedDataSets have two fields, one for input and one for the target.

__init__(*inp*, *target*)¶

Initialize an empty supervised dataset.

Pass *inp* and *target* to specify the dimensions of the input and target vectors.

__len__()¶

Return the length of the linked data fields. If no linked fields exist, return the length of the longest field.

addSample(*inp*, *target*)¶

Add a new sample consisting of *input* and *target*.

Batches(*label*, *n*, *permutation=None*)¶

Yield batches of the size of n from the dataset.

A single batch is an array of with dim columns and n rows. The last batch is possibly smaller.

If permutation is given, batches are yielded in the corresponding order.

Clear(*unlinked=False*)¶

Clear the dataset.

If linked fields exist, only the linked fields will be deleted unless *unlinked* is set to True. If no fields are linked, all data will be deleted.

Copy()¶

Return a deep copy.

Classmethod **loadFromFile**(*filename*, *format=None*)¶

Return an instance of the class that is saved in the file with the given filename in the specified format.

randomBatches(*label*, *n*)¶

Like .batches(), but the order is random.

Classmethod **reconstruct**(*filename*)¶

Read an incomplete data set (option arraysonly) into the given one.

saveToFile(*filename*, *format=None*, ***kwargs*)¶

Save the object to file given by filename.

splitWithProportion(*proportion=0.5*)¶

Produce two new datasets, the first one containing the fraction given by *proportion* of the samples.

9.4.2. Define the layers structure and 96oolean96

class pybrain.structure.modules.**LinearLayer**(*dim*, *name=None*)¶

Bases: **pybrain.structure.modules.neuronlayer.NeuronLayer**

The simplest kind of module, not doing any transformation.

__init__(*dim*, *name=None*)¶

Create a layer with dim number of units.

Class pybrain.structure.modules.**SigmoidLayer**(*dim*, *name=None*)¶

Bases: **pybrain.structure.modules.neuronlayer.NeuronLayer**

Layer implementing the sigmoid squashing function.

__init__(*dim*, *name=None*)¶

Create a layer with dim number of units.

9.4.3. Build a standard multilayer perceptron

pybrain.tools.shortcuts.**buildNetwork**(**layers*, ***options*)¶

Build arbitrarily deep networks.

Layers should be a list or tuple of integers, that indicate how many neurons the layers should have. *Bias* and *outputbias* are flags to indicate whether the network should have the corresponding biases; both default to True.

To adjust the classes for the layers use the *hiddenclass* and *outclass* parameters, which expect a subclass of **NeuronLayer**.

If the *recurrent* flag is set, a **RecurrentNetwork** will be created, otherwise a **FeedForwardNetwork**.

If the *fast* flag is set, faster arac networks will be used instead of the pybrain implementations.

9.4.4. Define the training algorithm

class pybrain.supervised.trainers.**BackpropTrainer**(*module*, *dataset=None*, *learningrate=0.01*, *lrdecay=1.0*, *momentum=0.0*, *verbose=False*, *batchlearning=False*, *weightdecay=0.0*)¶

Trainer that trains the parameters of a module according to a supervised dataset (potentially sequential) by backpropagating the errors (through time).

__init__(*module*, *dataset=None*, *learningrate=0.01*, *lrdecay=1.0*, *momentum=0.0*, *verbose=False*, *batchlearning=False*, *weightdecay=0.0*)¶

Create a BackpropTrainer to train the specified *module* on the specified *dataset*.

The learning rate gives the ratio of which parameters are changed into the direction of the gradient. The learning rate decreases by *lrdecay*, which is used to to multiply the learning rate after each training step. The parameters are also adjusted with respect to *momentum*, which is the ratio by which the gradient of the last timestep is used.

If *batchlearning* is set, the parameters are updated only at the end of each epoch. Default is False.

Weightdecay corresponds to the weightdecay rate, where 0 is no weight decay at all.

setData(*dataset*)¶

Associate the given dataset with the trainer.

testOnClassData(*dataset=None*, *verbose=False*, *return_targets=False*)¶

Return winner-takes-all classification output on a given dataset.

If no dataset is given, the dataset passed during Trainer initialization is used. If return_targets is set, also return corresponding target classes.

Train()¶

Train the associated module for one epoch.

trainEpochs(*epochs=1*, **args*, ***kwargs*)¶

Train on the current dataset for the given number of *epochs*.

Additional arguments are passed on to the train method.

trainOnDataset(*dataset*, **args*, ***kwargs*)¶

Set the dataset and train.

Additional arguments are passed on to the train method.

trainUntilConvergence(*dataset=None*, *maxEpochs=None*, *verbose=None*, *continueEpochs=10*, *validationProportion=0.25*)¶

Train the module on the dataset until it converges.

Return the module with the parameters that gave the minimal validation error.

If no dataset is given, the dataset passed during Trainer initialization is used. validationProportion is the ratio of the dataset that is used for the validation dataset.

If maxEpochs is given, at most that many epochs are trained. Each time validation error hits a minimum, try for continueEpochs epochs to find a better one.

9.5. Auxiliary python libraries

9.5.1. Math

Math Provides access to the mathematical functions defined by the C standard.
Math.**tanh**(*x*)

Return the hyperbolic tangent of *x*.

9.5.2. Random

Random implements pseudo-random number generators for various distributions.
Random.**shuffle**(*x*[, *random*])

Shuffle the sequence *x* in place. The optional argument *random* is a 0-argument function returning a random float in [0.0, 1.0); by default, this is the function **random()**. Note that for even rather small len(x), the total number of permutations of *x* is larger than the period of most random number generators; this implies that most permutations of a long sequence can never be generated.

9.5.3. Numpy

NumPy is the fundamental package for scientific computing with Python.

Numpy.exp(*x[, out]***)** = **<***ufunc 'exp'***>**

Calculate the exponential of all elements in the input array.

Parameters:

x *: array_like*

Input values.

Returns:

out *: ndarray*

Output array, element-wise exponential of *x*.

numpy.zeros(*shape*, *dtype=float*, *order='C'***)**

Return a new array of given shape and type, filled with zeros.

Parameters:

shape *: int or sequence of ints*

Shape of the new array, e.g., (2, 3) or 2.

Dtype *: data-type, optional*

The desired data-type for the array, e.g., numpy.int8. Default is numpy.float64.

order *: {'C', 'F'}, optional*

Whether to store multidimensional data in C- or Fortran-contiguous (row- or column-wise) order in memory.

Returns:

out *: ndarray*

Array of zeros with the given shape, dtype, and order.

Numpy.sum(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=False*)

Sum of array elements over a given axis.

Parameters:

a *: array_like*

Elements to sum.

Axis *: None or int or tuple of ints, optional*

Axis or axes along which a sum is performed. The default (*axis = None*) is perform a sum over all the dimensions of the input array. *Axis* may be negative, in which case it counts from the last to the first axis.

New in version 1.7.0.

If this is a tuple of ints, a sum is performed on multiple axes, instead of a single axis or all the axes as before.

Dtype *: dtype, optional*

The type of the returned array and of the accumulator in which the elements are summed. By default, the dtype of *a* is used. An exception is when *a* has an integer type with less precision than the default platform integer. In that case, the default platform integer is used instead.

Out *: ndarray, optional*

Array into which the output is placed. By default, a new array is created. If *out* is given, it must be of the appropriate shape (the shape of *a* with *axis* removed, i.e., numpy.delete(a.shape, axis)). Its type is preserved. See doc.ufuncs (Section "Output arguments") for more details.

Keepdims *: bool, optional*

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the original *arr*.

Returns:

sum_along_axis *: ndarray*

An array with the same shape as *a*, with the specified axis removed. If *a* is a 0-d array, or if *axis* is None, a scalar is returned. If an output array is specified, a reference to *out* is returned.

numpy.random.normal(*loc=0.0*, *scale=1.0*, *size=None***)**

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently, is often called the bell curve because of its characteristic shape (see the example below).

The normal distribution occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution.

Parameters:

loc *: float*

Mean ("centre") of the distribution.

Scale *: float*

Standard deviation (spread or "width") of the distribution.

Size *: int or tuple of ints, optional*

Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn. Default is None, in which case a single value is returned.

9.5.4. Datetime

The datetime module supplies classes for manipulating dates and times in both simple and complex ways.

Class datetime.**date**

An idealized naive date, assuming the current Gregorian calendar always was, and always will be, in effect. Attributes: **year**, **month**, and **day**.

9.5.5. Matplotlib

matplotlib.**rcParams**

An instance of **RcParams** for handling default matplotlib values.

Matplotlib.pyplot.**subplot**(**args*, ***kwargs*)

Return a subplot axes positioned by the given grid definition.

Keyword arguments:

axisbg:

The background color of the subplot, which can be any valid color specifier. See **matplotlib.colors** for more information.

Polar:

A 100oolean flag indicating whether the subplot plot should be a polar projection. Defaults to *False*.

Projection:

A string giving the name of a custom projection to be used for the subplot. This projection must have been previously registered.

matplotlib.pyplot.**scatter**(x, y, s=20, c=u'b', marker=u'o', cmap=None, norm=None, vmin=None, vmax=None, alpha=None,linewidths=None, verts=None, hold=None, **kwargs)

Make a scatter plot of x vs y, where x and y are sequence like objects of the same lengths.

Parameters:

x, y : array_like, shape (n,)

Input data

s : scalar or array_like, shape (n,), optional, default: 20

size in points^2.

C : color or sequence of color, optional, default

c can be a single color format string, or a sequence of color specifications of length **N**, or a sequence of **N** numbers to be mapped to colors using the **cmap** and **norm** specified via kwargs (see below). Note that **c** should not be a single numeric RGB or RGBA sequence because that is indistinguishable from an array of values to be colormapped. **C** can be a 2-D array in which the rows are RGB or RGBA, however.

Marker : **MarkerStyle**, optional, default: 'o'

See **markers** for more information on the different styles of markers scatter supports.

Cmap : **Colormap**, optional, default: None

A **Colormap** instance or registered name. **cmap** is only used if **c** is an array of floats. If None, defaults to rc **image.cmap**.

norm : **Normalize**, optional, default: None

A **Normalize** instance is used to scale luminance data to 0, 1. **Norm** is only used if **c** is an array of floats. If **None**, use the default **normalize()**.

Vmin, vmax : scalar, optional, default: None

vmin and **vmax** are used in conjunction with **norm** to normalize luminance data. If either are **None**, the min and max of the color array is used. Note if you pass a **norm** instance, your settings for **vmin** and **vmax** will be ignored.

Alpha : scalar, optional, default: None

The alpha blending value, between 0 (transparent) and 1 (opaque)

linewidths : scalar or array_like, optional, default: None

If None, defaults to (lines.linewidth,). Note that this is a tuple, and if you set the linewidths argument you must set it as a sequence of floats, as required by **RegularPolyCollection**.

Returns:

paths : **PathCollection**

matplotlib.pyplot.**xticks**(**args*, ***kwargs*)

Get or set the *x*-limits of the current tick locations and labels.

Matplotlib.pyplot.**yticks**(**args*, ***kwargs*)

Get or set the *y*-limits of the current tick locations and labels.

Matplotlib.pyplot.**grid**(b=None, which=u'major', axis=u'both', **kwargs)

Turn the axes grids on or off.

Matplotlib.pyplot.**show**(**args*, ***kw*)

Display a figure. When running in ipython with its pylab mode, display all figures and return to the ipython prompt.

In non-interactive mode, display all figures and block until the figures have been closed; in interactive mode it has no effect unless figures were created prior to a change from non-interactive to interactive mode (not recommended). In that case it displays the figures but does not block.

A single experimental keyword argument, *block*, may be set to True or False to override the blocking 102oolean102r described above.

9.6. R documentation

9.6.1. Nnet

Fit single-hidden-layer neural network, possibly with skip-layer connections.

Usage
nnet(x, …)

Arguments

formula	A formula of the form class ~ x1 + x2 + …
x	matrix or data frame of x values for examples.
Y	matrix or data frame of target values for examples.
Weights	(case) weights for each example – if missing defaults to 1.
Size	number of units in the hidden layer. Can be zero if there are skip-layer units.
Data	Data frame from which variables specified in formula are preferentially to be taken.
Subset	An index vector specifying the cases to be used in the training sample. (NOTE: If given, this argument must be named.)
na.action	A function to specify the action to be taken if Nas are found. The default action is for the procedure to fail. An alternative is na.omit, which leads to rejection of cases with missing values on any required variable. (NOTE: If given, this argument must be named.)
contrasts	a list of contrasts to be used for some or all of the factors appearing as variables in the model formula.
Wts	initial parameter vector. If missing chosen at random.
Mask	logical vector indicating which parameters should be optimized (default all).
Linout	switch for linear output units. Default logistic output units.
Entropy	switch for entropy (= maximum conditional likelihood) fitting. Default by least-squares.
Softmax	switch for softmax (log-linear model) and maximum conditional likelihood fitting. Linout, entropy, softmax and censoredare mutually exclusive.
Censored	A variant on softmax, in which non-zero targets mean possible classes. Thus for softmax a row of (0, 1, 1) means one example each of classes 2 and 3, but for censored it means one example whose class is only known to be 2 or 3.
Skip	switch to add skip-layer connections from input to output.
Rang	Initial random weights on [-rang, rang]. Value about 0.5 unless the inputs are large, in which case it should be chosen so thatrang * max(|x|) is about 1.
Decay	parameter for weight decay. Default 0.
Maxit	maximum number of iterations. Default 100.
Hess	If true, the Hessian of the measure of fit at the best set of weights found is returned as component Hessian.
Trace	switch for tracing optimization. Default TRUE.
MaxNWts	The maximum allowable number of weights. There is no intrinsic limit in the code, but increasing MaxNWts will probably allow fits that are very slow and time-consuming.
Abstol	Stop if the fit criterion falls below abstol, indicating an essentially perfect fit.
Reltol	Stop if the optimizer is unable to reduce the fit criterion by a factor of at least 1 – reltol.

… arguments passed to or from other methods.

9.6.2. RSNNS

The Stuttgart Neural Network Simulator (SNNS) is a library containing many standard implementations of neural networks. This package wraps the SNNS functionality to make it available from within R.

As the high-level api is already quite powerful and flexible, you'll most probably normally end up using one of the functions: **mlp**, dlvq, rbf,rbfDDA, elman, jordan, som, art1, art2, artmap, or assoz, with some pre- and postprocessing. These S3 classes are all subclasses ofrsnns. For this project the mlp function has been used.

9.6.2.1. mlp

This function creates a multilayer perceptron (MLP) and trains it. MLPs are fully connected feedforward networks, and probably the most common network architecture in use. Training is usually performed by error backpropagation or a related procedure.

Usage

mlp(x, …)

Arguments

x	a matrix with training inputs for the network
…	additional function parameters (currently not used)
y	the corresponding targets values
size	number of units in the hidden layer(s)
maxit	maximum of iterations to learn
initFunc	the initialization function to use
initFuncParams	the parameters for the initialization function
learnFunc	the learning function to use
learnFuncParams	the parameters for the learning function
updateFunc	the update function to use
updateFuncParams	the parameters for the update function
hiddenActFunc	the activation function of all hidden units
shufflePatterns	should the patterns be shuffled?
linOut	sets the activation function of the output units to linear or logistic
inputsTest	a matrix with inputs to test the network
targetsTest	the corresponding targets for the test input
pruneFunc	the pruning function to use
pruneFuncParams	the parameters for the pruning function. Unlike the other functions, these have to be given in a named list.

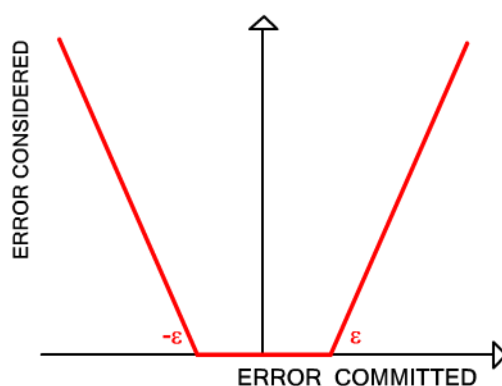
9.7. Futures paths to follow

9.7.1. Support vector regression

After analysing the performance of the neural network algorithm, some alternatives such as the support vector machines, proof to be a robust alternative. However, support vector machines have been used to solve classification problems rather than regression problems.
To overcome this fact, the most feasible approach is to transform the data, from a continuous magnitude to a discrete magnitude –with a finite number of classes.
Another approach, would be adapting the algorithm to output a continuous response within a given range of error.

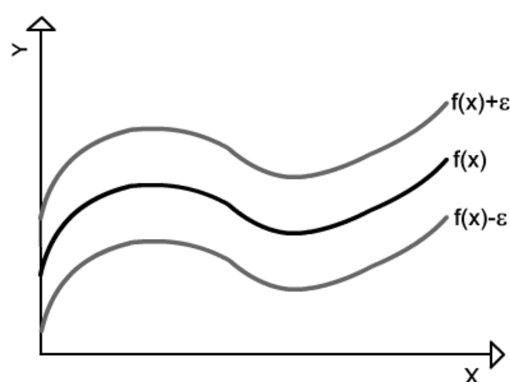
9.7.2. How it Works?

To perform the optimization of the support vector regression the error is defined by the function plotted below:



Plot 47. Error plot

This error function determines three main regions: a negative region (overestimation), a positive region (underestimation) and a non-error region. When a prediction is performed during the training, the result can be classified in any of these three intervals. The values fitted in the non-error region, don't generate any correction on the margins creating a tube tolerant area:



Plot 48. Estimation regions

Coming back to a mathematical point of view, this problem can be defined as a classical optimization problem. This quadratic optimization problem will be defined as the minimization of the margin, and a certain cost function:

$$\min_{w,b} \frac{1}{2} w^T \cdot w + C \sum_{i=1}^{l} (\xi_i + \xi_i^*)$$

To minimize this value, some restrictions have to be applied to the value of the errors:

$$s.t. \begin{cases} y_i - (w^T \cdot \phi(x) + b) < \varepsilon + \xi_i \\ (w^T \cdot \phi(x) + b) - y_i \le \varepsilon + \xi_i^* \\ \xi_i, \xi_i^* \ge 0, i = 1..l \end{cases}$$

$\phi(x)$ Kernel function
w Margin
ε Threshold
ξ Tolerance error

Once the SVR is trained, the result is generated following the formula:

$$f(x) = \sum_{i=1}^{l} \theta_i \phi(x, x_i) + b$$

9.7.3. SVR over sklearn

The mathematical problem can be solved using calculus packages such as scikit-learn (built on NumPy, SciPy and matplotlib). In this case, the function that implements this algorithm is defined below:

class sklearn.svm.**SVR**(*kernel='rbf'*, *degree=3*, *gamma=0.0*, *coef0=0.0*, *tol=0.001*, *C=1.0*, *epsilon=0.1*, *shrinking=True*,*cache_size=200*, *verbose=False*, *max_iter=-1*)

Epsilon-Support Vector Regression.
The free parameters in the model are C and epsilon.
The implementation is based on libsvm.

Parameters:

C : float, optional (default=1.0)
Penalty parameter C of the error term.

Epsilon : float, optional (default=0.1)
Epsilon in the epsilon-SVR model. It specifies the epsilon-tube within which no penalty is associated in the training loss function with points predicted within a

distance epsilon from the actual value.

Kernel : string, optional (default='rbf')

Specifies the kernel type to be used in the algorithm. It must be one of 'linear', 'poly', 'rbf', 'sigmoid', 'precomputed' or a callable. If none is given, 'rbf' will be used. If a callable is given it is used to precompute the kernel matrix.

Degree : int, optional (default=3)

Degree of kernel function is significant only in poly, rbf, sigmoid.

Gamma : float, optional (default=0.0)

Kernel coefficient for rbf and poly, if gamma is 0.0 then 1/n_features will be taken.

Coef0 : float, optional (default=0.0)

independent term in kernel function. It is only significant in poly/sigmoid.

Shrinking: 107oolean, optional (default=True) :

Whether to use the shrinking heuristic.

Tol : float, optional (default=1e-3)

Tolerance for stopping criterion.

Cache_size : float, optional

Specify the size of the kernel cache (in MB).

Verbose : bool, default: False

Enable verbose output. Note that this setting takes advantage of a per-process runtime setting in libsvm that, if enabled, may not work properly in a multithreaded context.

Max_iter : int, optional (default=-1) Hard limit on iterations within solver, or -1 for no limit.

Attributes:

support_ : array-like, shape = [n_SV]

Indices of support vectors.

Support_vectors_ : array-like, shape = [nSV, n_features]

Support vectors.

Dual_coef_ : array, shape = [1, n_SV]

Coefficients of the support vector in the decision function.

Coef_ : array, shape = [1, n_features]

Weights assigned to the features (coefficients in the primal problem). This is only available in the case of linear kernel.

Coef_ is readonly property derived from dual_coef_ and support_vectors_.

Intercept_ : array, shape = [1]

Constants in decision function.

To analyse the behaviour of this algorithm, the 5 main parameters of the SVR constructor are set in different values. The objective is to generate all the possible combinations, so as to view the interaction between the algorithm variables and the result (ARV and RMSE).

```
Result = []
for c in [0.1, 1, 10]:
    for e in [0.00001, 0.0001, 0.001]:
        for k in [“rbf”, “linear”, “poly”, “sigmoid”]:
```

```
            for d in [1, 2, 3, 4]:
                for g in [0.01, 0.1, 1.0]:
                    clf = sklearn.svm.SVR(C = c, epsilon = e, kernel = k,
degree = d, gamma = g)
                    clf.fit(trX, trY)
                    pred = clf.predict(teX)
                    rmse = (pred - teY).std()
                    arv = np.sum((teY - pred)**2)/np.sum((teY -
teY.mean())**2)
                    result.append([c, e, k, d, g, rmse, arv])

result.sort(key=lambda x: x[5])
for line in result:
    print line
```

Part of the output is shown below

```
1 0.0001 rbf 1 0.1 0.0606960337886 0.186853286507
1 1e-05 rbf 1 0.1 0.0607139028994 0.186916570073
1 0.001 rbf 1 0.1 0.0607431068359 0.187157339248
10 0.001 poly 2 1.0 0.0607443046984 0.186746362622
10 0.001 poly 2 0.1 0.0607632618916 0.186859954353
…
```

The best result, after checking all the combinations, is a SVR applying a RBF with a C = 1 and an epsilon = 0,0001. Reducing the epsilon an order of magnitude doesn't seem to improve the performance. The value of epsilon is reasonable if we consider that the range of values goes from 0 to 1, and the tolerated error is 0,01%.