National College of Ireland

Configuration Manual

MSc Research Project
Data Analytics

Stephane Nichanian
Student ID: 18202632

School of Computing
National College of Ireland

Supervisor: Dr. Rashmi Gupta

National College of Ireland
Project Submission Sheet
School of Computing

National College of Ireland

Student Name:	Stephane Nichanian
Student ID:	18202632
Programme:	Data Analytics
Year:	2018
Module:	MSc Research Project
Supervisor:	Dr. Rashmi Gupta
Submission Due Date:	21/09/2020
Project Title:	Configuration Manual
Word Count:	2283
Page Count:	11

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	
Date:	27th September 2020

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

Configuration Manual

Stephane Nichanian
18202632

1 Introduction

This document aims to provide the hardware and software information used to complete this project. It also details and explain the code used to generate the various models, predictions, statistics and graphics that compose the project called : **"Understanding the impact of COVID-19 on electrical demand"**
Due to the length of the code and the forecasting methodology, the coding has been separated in 4 different files: pre-processing, dynamic forecast update, regression and plots.

2 System Configuration

2.1 Hardware

The hardware used for this project is a Acer Helios 300 laptop with the following configuration:

- Processor: Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz,
- Installed memory (RAM): 16.0 GB,
- System Type: Windows OS, 64-bit,
- GPU: NVIDIA GeForce GTX 1060,
- Storage: 275 GB SSD,

2.2 Software

Microsoft Excel 2016: the popular Microsoft data file management tool has been used to store all the data downloaded from various websites as well as to store the data generated by the code in R studio. The extension used by the Excel program is in .csv format which is the preferred format to import and export data in R studio.

R studio: This open source tool provides an user friendly environment to code in R language. Multiple functionalities are offered such as automatic code error detection, function auto-completion and a visual environment for graph generation. The version of R Studio used for this project is Version 1.3.1056.

3 Data pre-processing

This Chapter explains how the original dataset and predictor variables were obtained and cleaned.

3.1 Original dataset

The raw dataset containing electrical consumption information is obtained from Reseaux Transport Electricite RTE website, https://www.rte-france.com/. This company is responsible for the transportation of all electricity in France from production sites to final usage location. They are therefore also responsible for building and maintaining the entirety of the electrical grid in France along with electricity demand monitoring. Part of a recent transparency scheme, RTE has made available all electrical data in all of the regions of France. The data obtained from the RTE website, therefore contains the quarter hourly electrical demand readings for the region of Ile-De-France from the 1st of January 2016 to the 29th June 2020.
The following code snipet figure 1 shows how the separate data files were merged in one file. The R library used for the data pre-processing is "tidyverse" which is an ensemble of libraries necessary for basic data cleaning and transformation tasks in R. The following snippet shows the corresponding code.

```r
#Each file is given separately, per year
France_2016 <- read.csv("RTE_Ile-de-France_2016.csv", header = T, na.strings = "N/A", stringsAsFactors = F)
France_2017 <- read.csv("RTE_Ile-de-France_2017.csv", header = T, na.strings = "N/A", stringsAsFactors = F)
France_2018 <- read.csv("RTE_Ile-de-France_2018.csv", header = T, na.strings = "N/A", stringsAsFactors = F)
France_2019 <- read.csv("RTE_Ile-de-France_En-cours.csv", header = T, na.strings = "N/A", stringsAsFactors = F)
France_2020 <- read.csv("RTE_Ile-de-France_En-cours-TR.csv", header = T, na.strings = "N/A", stringsAsFactors = F)

#We only need the columns that have the date, hour and electricity consumption values
France_2016 <- France_2016[,3:5]
France_2017 <- France_2017[,3:5]
France_2018 <- France_2018[,3:5]
France_2019 <- France_2019[,3:5]
France_2020 <- France_2020[,3:5]

#We join all the separate Data Frames (DF) in one
France <- rbind.data.frame(France_2016,France_2017,France_2018,France_2019,France_2020)
rownames(France) <- 1:nrow(France)
```

Figure 1: CSV files merging

Each file is loaded separately using read.csv() function followed by the combination of each file in data frame format by row using r.bind.data.frame().
The quarterly and half hourly values are deleted by subsetting values ending with 15, 30 and 45 (see figure 2. It is also important to note that the time and date column must be transformed into a time format variables, which makes the time related data transformation and visualisation much easier. This is done using the strptime() function whilst indicating the format of the existing time variable in the arguments. The following snippet 2 demonstrate this important step along with the full hourly data transformation.

```r
#We convert the newly formed column into a date format for easier usage
France$TimeHour <- strptime(France$TimeHour, format="%m/%d/%Y - %H:%M", tz = "CET")

#We will now omit the quarter and half-hourly observations.
#As we simply do not require this level of detail.
France <- (subset(France, format(TimeHour,"%M") != 15 & format(TimeHour,"%M") != 45))

#Similaryl all the observation have half hourly readings.
France <- (subset(France, format(TimeHour,"%M") != 30))
```

Figure 2: Full hour transformation and time conversion code

The missing values are handled using the moving average method, which uses the before and after available observations and averages them. For this task the library used is "imputeTS" which is a library containing multiple functions (mean, linear interpolation, Last Observation Carried Forward LOCF) to compute missing values in a time series. The following snippet 3 shows the snippet for the missing values and the library used. na_ma is the moving average method.

```r
#We will use the imputeTS package for this
library(imputeTS)

France$Electricity <- na_ma(France$Electricity, weighting = "simple")
```

Figure 3: Missing values using moving average method

It should also be noted that, there were additional missing values in the windspeed and UVindex variables which were replaced by moving average and last observation carried forward method respectively.

3.2 Weather variables

The weather variables are imported from the Dark Sky API.
The R libraries necessary for this section are: httr for the GET request, jsonlite to deal with the JSON format received by the API and stringr to link the different items of the API URL. The snippet in figure 4 shows initialization code for the API settings.

```r
#the Unix time in Ile de France on January 1st 2016 at midnight is:
Start_Time <- 1451602800

#Setting all the parameters for our API calls.
#The longitude and latitude specified here are for Paris.
darksky_url <- "https://api.darksky.net/forecast/xxxxxxxx/"
longitude_IDF <- "48.8566,"
latitude_IDF <- "2.3522,"
units_darksky <- "?units=si"

#We are now creating a vector (in UNIX timebase) from
#January 1st to June 29th 2020 (total of 1642 days)
#Each day at midnight will be one item in the vector.
Time_vec <- rep(0,1642)
Time_vec[1] <- Start_Time
j=1

for (i in 1:1641) {
  Time_vec[j+1] <- Time_vec[j] + 86400
  j=j+1
}
```

Figure 4: DarkSky API initialisation parameters

The time stamp is in UNIX format and therefore the initial date of January 1st 2016 at midnight in "CET" timezone becomes 1451602800 in UNIX format. The darksky url is added along with the user API key (replaced by xxxxxx in this snippet as this is a paying service). The longitude and latitude of Paris have been specified as this is the most densely populated area of Ile-De-France and will have the highest electrical demand. The following step is to create a time vector called Time_vec in figure 4, that contains UNIX time values for each day at midnight ranging from January 1st 2016 to June 29th 2020. There are 86400 seconds in one day, therefore adding this number to each previous value will represent a new day.
After initializing the setting, we can now call the API and store the information in data frames as shown in figure 5.

```
#Creating the API call URL with all our settings string together
darksky_call <- str_c(darksky_url,
                      longitude_IDF, latitude_IDF, Time_vec, units_darksky )

#The following loop requests the API weather information for each day
weather_DF <- list()
j=1
for (i in 1:length(Time_vec)) {
  request <- GET(darksky_call[j])

  #Encoding the content in a simple text format
  response <- content(request, as = "text", encoding = "UTF-8")

  #Storing the information in a DF.
  weather <- fromJSON(response, flatten = TRUE)
  weather_DF[[j]] <- data.frame(weather[5])
  j= j+1
}
```

Figure 5: DarkSky API call loop

The API call link is created by stringing together the various components of the URL . We then create a loop that iterates through each day (coded in UNIX time in $Time_vec$ variable) and places a separate request for each day. The Darsky API returns the weather information, in JSON format and UTF-8 coding, for each day (which is contain in the fifth item of the response variable). We then store this information in a data frame called $Weather_DF$.
It should be noted at this point that the first 1000 API requests are free, however any call made after that is charged at 0.0001$ per call. Therefore, the data is stored into a local file and any other task done after this point loads the local file instead of creating a new API call.
Due to the lengthy pre-processing task for this project, the method of storing files locally and re-loading them is used multiple times through the code, The following code snippet displays how the local file is saved and reloaded for processing.

```
write.csv(DF,"DF3.csv", row.names = FALSE)

#Setting the directory where all files will be used from for this project
setwd("C:\\Users\\Stephane\\Documents\\Uni docs\\Dissertation\\Dataset\\RTE\\DF")

library(tidyverse)

#Load data from here
DF <- read.csv("DF3.csv", stringsAsFactors = TRUE)

#We convert TimeHour into a date format for easier use
DF$TimeHour <- strptime(DF$TimeHour,format='%Y-%m-%d %H:%M:%S')
DF$Month <- as.factor(DF$Month)
```

Figure 6: Local file saving and loading

write.csv() and read.csv() are used to respectively write and store the data frame in a csv file, whilst setwd() is used to set the directory.

3.3 Time variables

The time related variables must be coded manually and added to the dataset. The snippet in figure 7 shows how the day of the week is added. This is done by using the weekdays() function which takes as input a date (at the correct date format) and returns the corresponding day of the week. This is then added as a multiple level factor in the data frame.

```
#We will now add the work days and saturday and sundays as separate items.
DF$Workday <- NA
Weekdays_vec <- weekdays(DF$TimeHour)

work_day <- c("Monday","Tuesday","Wednesday","Thursday","Friday")
Sat <- "Saturday"
Sun <- "Sunday"
j = 1

for (i in 1:nrow(DF)){

  if(Weekdays_vec[j] %in% work_day) {
    DF$Workday[j] <- "Workday"

  } else if (Weekdays_vec[j] %in% Sat) {
    DF$Workday[j] <- "Saturday"

  } else if (Weekdays_vec[j] %in% Sun) {
    DF$Workday[j] <- "Sunday"
  }
  j = j+1
}
```

Figure 7: Weekday code

The remaining of time related components are coded in a similar way.

3.4 Lagged demand and temperature variables

The lagged temparature variables are also coded manually. The snippet in figure 8 demonstrates how the minimum temperature is added to the data frame.

```
j = 1
for ( i in 1:nrow(DF)){
  if (j < 26){
    DF$Temp_min[j] <- min(DF[1:j,]$Temperature)

  } else {
    DF$Temp_min[j] <- min(DF[(j-25):j-1,]$Temperature)
  }
  j = j + 1
}
```

Figure 8: Minimum temperature

We simply create a loop that iterates through each row and checks for the last 23 available temperatures and selects the minimum value.
Coded similarly, the lagged temperature variables contain the following variables: temperature 24 hours ago, temperature 48 hours ago, minimum temperature in last 24 hours, maximum temperature 24 hours ago, average temperature in last 7 days.

The lagged demand variables are then created. For example, figure 9 shows the code to create the last observed electricity demand 24 hours ago.

```
#Same Period 1 day before
j = 1

for ( i in 1:nrow(DF)){
  if (j < 25){
    DF$Temp_d1[j] <-  DF$Temperature[j]

  } else {
    DF$Temp_d1[j] <-  DF$Temperature[j-24]
  }
  j = j + 1
}
```

Figure 9: Last electrical value, 24 hours ago

Coded similarly, the lagged demand variables contain the following variables: demand 24 hours ago, demand 48 hours ago, minimum demand in last 24 hours, maximum demand 24 hours ago, average demand in last 7 days.
In figure 10 we demonstrate that the dataframe does not contain any missing values, using the sapply() function combined with the is.na() function that return true when an observation is NA.

```
> sapply(DF,function(x) sum(is.na(x)))
       Dates     TimeHour         Hour  Temperature     Humidity    Windspeed      UVIndex  Electricity      Weekday
           0            0            0            0            0            0            0            0            0
     Holiday Bank_Holiday        trend
           0            0            0
```

Figure 10: Missing values check

3.5 Dynamic forecast data update

The dynamic forecast update is implemented by calling a different R script, every time a prediction is made. This script updates the lagged variables and replaces them in the existing data frame. The snippet in figure 11 demonstrates the call to the dynamic update script from the main R code.

```r
#Call the script that will update the lagged demands with the newly calculated forecast values
source("C:\\Users\\Stephane\\Documents\\Uni docs\\Dissertation\\Code\\Dynamic forecast update.R")
```

Figure 11: Dynamic update script call

4 Predictive Models

4.1 General Additive Model (GAM)

Figure 12 shows the GAM model in R. The main function is gam() obtained from the "mgvc" package created by Simon Wood a professor of Statistics at Bristol University and author of multiple books on statistics and generalized additive models. Figure 12 shows the code to create the GAM model at midnight.

```r
if ( i < 25){
  if (DF_test$Hour[i] == "0:00"){
  gam0 <- gam(Electricity ~  s(Trend, bs= "ps") + Month + Weekday + Holiday + Bank_Holiday +
                s(Temperature, bs= "ps") + s(Temp_diff, bs= "ps") + s(Temp_max, bs= "ps") + s(Temp_min, bs= "ps") +
                s(Temp_d1, bs= "ps") + s(Temp_d2, bs= "ps") + s(Temp_avg, bs= "ps") +
                s(Demand_d1, bs= "ps") + s(Demand_d2, bs= "ps") + s(Demand_max, bs= "ps") +
                s(Demand_min, bs= "ps") + s(Demand_avg, bs= "ps") + Month*Temperature,
                data = subset(DF_train, Hour == DF_test$Hour[i]), select = TRUE)
```

Figure 12: GAM model in R

As explained in the methodology chapter of the main report, there are 24 different GAM models which each represent one hour of the day. In the R code, this is done by creating a loop that iterates through each observation of the testing set and creates the GAM model for the specific hour of the observation.
The response variable is Electricity which follows a Gaussian distribution and a logarithmic link function. The Holiday and Bank Holiday predictors are categorical factors and as such do not require to be integrated in smoothing function and are left in their original form. The terms contained in the s() function are all transformed via smoothing functions. We note that the trend evolution is not linear, therefore we include the trend variable in the smoothing function s(). Similarly, the lagged demand and lagged temperature terms which all display non-linear behaviour are also included in smoothing functions. The method bs = "ps" indicates that the Smooth Splines method has been used.
Finally, the select = TRUE argument is used for feature selection and eliminates any variable that does not contribute to the model.
A manual feature selection process was also implemented to understand the contribution of lagged temperature and lagged demand variables. Three separate models were created with a subset of variables. The code in figure 13 demonstrates all three models as coded in the project, along with the ANOVA test to create the comparison statistics.

```r
#We compute the residuals and statistics about each model.
gam_complete <- gam(log10(Electricity) ~ s(Trend, bs= "ps") + Month + Weekday + Holiday + Bank_Holiday +
                          s(Temperature, bs= "ps") + s(Temp_diff, bs= "ps") + s(Temp_max, bs= "ps") +
                          s(Temp_min, bs= "ps") + s(Temp_d1, bs= "ps") + s(Temp_d2, bs= "ps") +
                          s(Temp_avg, bs= "ps") +
                          s(Demand_d1, bs= "ps") + s(Demand_d2, bs= "ps") + s(Demand_max, bs= "ps") +
                          s(Demand_min, bs= "ps") + s(Demand_avg, bs= "ps") + Month*Temperature,
                        data = subset(DF_train, Hour == "22:00"), select = TRUE)

gam_no_lag_temp <- gam(log10(Electricity) ~ s(Trend, bs= "ps") +  Month + Weekday + Holiday + Bank_Holiday +
                         s(Temperature, bs= "ps") +
                         s(Demand_d1, bs= "ps") + s(Demand_d2, bs= "ps") + s(Demand_max, bs= "ps") +
                         s(Demand_min, bs= "ps") + s(Demand_avg, bs= "ps") + Month*Temperature,
                       data = subset(DF_train, Hour == "22:00"), select = TRUE)

gam_no_lag_dmd <- gam(log10(Electricity) ~  s(Trend, bs= "ps") + Month + Weekday + Holiday + Bank_Holiday +
                        s(Temperature, bs= "ps") + s(Temp_diff, bs= "ps") + s(Temp_max, bs= "ps") +
                        s(Temp_min, bs= "ps") + s(Temp_d1, bs= "ps") + s(Temp_d2, bs= "ps") +
                        s(Temp_avg, bs= "ps") + Month*Temperature,
                      data = subset(DF_train, Hour == "22:00"), select = TRUE)
```

Figure 13: GAM models for feature testing

We can see all three models created at 22h with the subsetted predictor variables.

Once the model is created for the specific hour, the prediction is made on the testing set using the predict() function, the resulting forecast is then added into the original dataframe in the "Forecast" column. The electricity variable is brought to the power of 10 due to the log10 transformation done for the GAM model. See Figure 14

```r
#Hourly prediction
DF_Pred  <- predict(gam_0 ,DF_test[i,] )
#Replace forecast column by the values we just forecasted (1 value at a time)
DF$Forecast[as.integer(rownames(DF_test[i,]))] <- 10^(as.numeric(DF_Pred))
```

Figure 14: Model prediction

4.2 GAM assumptions

Once the model is created, the GAM assumptions are checked by generating visuals of the residuals using the gam.check() function from the mgvc package in R. This method returns all the standard graphs namely: histogram of residuals, residual vs linear predictor, response vs fitted values and QQplot. See figure 15 for the corresponding code. The par 2x2 indicates the number of graphs to display in the output window.

```r
#Gam assumption plots
par(mfrow = c(2,2))
gam.check(gam0)
```

Figure 15: Residual plot code

The Durbin-Watson (DW) statistic is computed using the durbinWatsonTest() function from forecast package. This function returns the DW test statistic as well as p-value for the test. The argument for this function is the residuals of the GAM model.

The AIC and R squared statistic used to compare the performance of nested GAM models are generated by applying a ANOVA method to the GAM models. Figure 13 demonstrates the code for the ANOVA function.

5 ARIMA Regression

The second model ARIMA Regression is created using the auto.arima() function from the forecast package. The predictor variables (called regressors in this model) are passed in the argument of the auto.arima() function. The code to create the ARIMA regression model for midnight is shown in figure 16.

```
if (DF_test$Hour[i] == "0:00"){
  fit0 <- auto.arima(DF_train_hour[, "Electricity"], xreg = xreg)
```

Figure 16: Model creation

Similarly to the GAM model, one model is created for each hour of the day through a loop that iterates through each observation and subsets the corresponding hour data. The prediction is done by using the forecast() function from the forecast package. The regressors for the corresponding observation are added in the arguments and prediction is made using the model just created. Figure 17 shows the prediction code

```
if (DF_test$Hour[i] == "0:00"){
  fcast <- forecast(fit0,xreg =t(xreg_test[i,]) )
```

Figure 17: Model prediction

5.1 Arima regressors

The ARIMA regressors are the same as the GAM regressors. However the auto.arima() function requires categorical variables to be in 0 and 1 format. Figure 18 shows the one hot encoding code to transform categorical variables into binary variables.

```
xreg_test <- one_hot(as.data.table(xreg_test))
xreg_test <- as.data.frame(xreg_test)
```

Figure 18: One hot encoding

The function one_hot() from the mltools package in R is used and subsequently transformed into data frame format.
Finally, the ACF plots to graphically verify residuals auto-correlation is with the Acf() function from the forecast package.

6 Evaluation

The training and testing set are manually set to the dates chosen by the user. As specified in the Evaluation section of the main report, we first create the model on 5 single day dates spread throughout the year, we then create a 3-month testing set and finally another 3-month set during lockdown. The code in figure 19 demonstrates the training and testing periods.

```r
#Period 1. January 1-2 2019
DF_train <- DF[1:26254,]
DF_test <- DF[26255:26303,]

#Period 2. March 15-16 2019
DF_train <- DF[1:28006,]
DF_test <- DF[28007:28055,]

#Period 3. May 30-31 2019
DF_train <- DF[1:29828,]
DF_test <- DF[29829:29877,]

#Period 4. July 15-16 2019
DF_train <- DF[1:30932,]
DF_test <- DF[30933:30981,]

#Period 5. October 1-2 2019
DF_train <- DF[1:32803,]
DF_test <- DF[32804:32827,]



#Training is from January 1st 2016 to March 16th 2019
#Testing is from march 17th 2019 to June 29th 2019
#(same as the actual validation period, one year before)
DF_train <- DF[1:28054,]
DF_test <- DF[28055:30572,]


#Application for COVID-19 is from march 17th 2020 to June 29th 2020
#(same as the actual validation period, one year before)
DF_train <- DF[1:36837,]
DF_test <- DF[36838:39355,]
```

Figure 19: Training vs testing of all cases

As we can see the training and testing dates are specified by indexing the row of the main data frame which corresponds to the desired date (at midnight).

The evaluation methodology for this project is done through Mean Absolute Percentage Error (MAPE) and graphs. Once the model is created and the testing set values are forecasted, we store all variables in a data frame and compute hourly and daily MAPE statistics. Figure 20 shows the corresponding code.

```r
#Creating the results data frame
results <- cbind.data.frame(DF$Dates, DF$Hour, DF$Electricity, DF$Forecast)
results <- results[28055:30572,]

rownames(results) <- c(1:nrow(results))
colnames(results) <- c("Date", "Hour", "Electricity", "Forecast")

results$MAPE <- NA
results$MAPE_Daily <- NA

for ( i in 1:nrow(results)){
  results$MAPE[i] <- accuracy(results$Forecast[i], results$Electricity[i])[5]

  if (as.character(results$Hour[i]) == "23:00"){
    results$MAPE_Daily[i] <- mean(results$MAPE[(i-23):i])

  }
}

write.csv(results, "results AR.csv", row.names = FALSE)
```

Figure 20: Results dataframe

The MAPE statistic is computed using the accuracy() function from the forecast

packege. The MAPE statistic is returned in the fifth position of the accuracy function. The daily MAPE is computed by calculatin the mean of each hourly observation when 23h is reached.

The graphs in the evaluation section are composed of multiple types such as bar graphs, line graphs or scatterplots. The ggplot2 package is used to generate all the graphs. Figure 21 shows the code used to generate the bar graph of Daily percentage change during lockdown.

```
#Barplot of difference from lockdown and no lockdown
ggplot(DF_Daily,aes(x=Date,y=Percentage)) +
  geom_bar(stat="identity", fill = "orange") +
  ylab("Percentage Difference") + xlab("Date") +
  theme(plot.title = element_text(hjust = 0.5)) +
  ggtitle("Daily percentage change in lockdown")
```

Figure 21: Results dataframe

The aes argument captures the content of the x and y axis, in this case the time variable is in the x axis and the percentage change is in the y axis. The dataframe considered for the plot is called DF_Daily, the geom_bar function indicates the usage of a barplot, the lab() functions allow us to rename the x and y axis and ggtitle() adds a title to the plot.