Final Project

Master in Energy Engineering (MUEE)

Artificial Intelligence Applied to Demand Forecasting

UPC

July 1, 2022

Author:	Óscar Cabrera Redondo
Director:	Mónica Aragüés Peñalba
Director:	Antonio Emmanuel Saldaña González
Announcement:	July 1, 2022

ETSEIB

Escola Tècnica Superior
d'Enginyeria Industrial de Barcelona

UPC

1 Glossary

- ACF: Autocorrelation Function.
- AI: Artificial Intelligence.
- ANN: Artificial Neural Network.
- AR: Autoregressive Model.
- BPTT: Backpropagation Through Time.
- BRNN: Bidirectional Recurrent Neural Networks.
- CNN: Convolutional Neural Network.
- DSM: Demand Side Management.
- ETS: Error, Trend and Seasonality. Acronym for Exponential Smoothing.
- ETSEIB: Escola Tècnica Superior d'Enginyeria Industrial de Barcelona.
- GRUs: Gated Recurrent Units.
- HVAC: Heating, Ventilation, and Air-Conditioning.
- LCA: Life Cycle Assessment.
- LSTM: Long Short-Term Memory.
- MA: Moving Average.
- MAE: Mean Absolute Error.
- NN: Neural Network.
- PACF: Partial Autocorrelation Function.
- REE: Red Eléctrica España.
- ReLu: Rectified Linear Unit.
- RMSE: Root Mean Square Error.
- RNN: Recurrent Neural Network.
- TEG: Triethylene Glycol.
- UPC: Universitat Politècnica de Catalunya.
- VRE: Variable Renewable Energy.

ETSEIB

Contents

1 Glossary **1**

2 Preface **7**

3 Introduction **11**
3.1 Objective 11
3.2 Scope of the project 11

4 State of the art **13**
4.1 Statistical methods 15
4.1.1 Exponential smoothing model 15
4.1.2 SARIMA model 18
4.1.3 SARIMAX model 20
4.2 Neural Networks 21
4.2.1 Convolutional Neural Networks (CNN) model 22
4.2.2 Recurrent Neural Network (RNN) model 23

5 Methodology **27**
5.1 Data to analyse 27
5.2 Data treatment 29
5.2.1 Cleaning data 29
5.2.2 Data treatment for neural networks 32
5.3 Algorithms 34
5.3.1 Naive model 34
5.3.2 Exponential smoothing model 43
5.3.3 SARIMA model 47
5.3.4 SARIMAX model 53
5.3.5 Convolutional Neural Network (CNN) model 55
5.3.6 Recurrent Neural Network (RNN) univariate model 59
5.3.7 Recurrent Neural Network (RNN) multivariate model 62
5.3.8 Recurrent Neural Network (RNN) pandemic prediction 65
5.4 Results 66
5.5 Comparison 68

Conclusions **71**

Acknowledgements **73**

A Economic Analysis **77**

B Environmental Impact **79**
B.1 Goal and scope 79
B.2 Inventory and Analysis 79
B.3 Impact Analysis 81
B.4 Interpretation 82

C Data treatment Code **85**

D Naive Code 87

E Exponential Smoothing Code 91

F SARIMA Code 95

G SARIMAX Code 99

H CNN Code 101

I CNN-LSTM Code 105

J LSTM multivariate Code 109

List of Figures

1 Results taken from Jason Brownlee [1] 14
2 Weighted linear sum 16
3 Box-Cox transformation 18
4 Nested concepts 21
5 Neural nodes information transmission 22
6 RNN cell 23
7 LSTM cell architecture 25
8 London shape 28
9 Canarias shape 29
10 RMSE and split code 36
11 Walk forward validation 36
12 Score model code 37
13 Grid search code 38
14 Prediction of energy demand in London (persist method) 40
15 Prediction of energy demand in London (mean method) 41
16 Prediction of energy demand in Canarias (persist method) 42
17 Prediction of energy demand in Canarias (mean method) 43
18 Exponential smoothing in London scenario 46
19 Exponential smoothing in Canarias scenario 47
20 Autocorrelation function for the London scenario 49
21 Partial Autocorrelation Function for the London scenario 50
22 SARIMA forecast in London scenario 51
23 ACF and PAFC 51
24 SARIMA in Canarias scenario 53
25 SARIMAX forecast for London scenario 54
26 SARIMAX forecast for Canarias scenario 54
27 CNN prediction in London scenario 58
28 CNN prediction in Canarias' scenario 59
29 CNN-LSTM forecast in London scenario 61
30 CNN-LSTM forecast in Canarias scenario 62
31 Input variables of an LSTM forecast 63
32 LSTM for multivariate Canarias' scenario 65
33 Canarias scenario prediction in during lockdown 66
34 RMSE London scenario 69
35 RMSE Canarias scenario 69
36 Difference between the RMSE of both scenarios 70
37 Report flow 80
38 Report flow 81
39 Aquatic ecotoxicity 81
40 Global Warming Effect 82
41 Non-renewable energy 82

2 Preface

As the actual grid must be improved and integrate variable renewable energy (VRE) into it, different methods and strategies have been developed. These techniques permit higher flexibility, better stability and, therefore, a larger penetration of renewable energy in the grid.
An important technique to make this happen is demand forecasting. This technique is very common and there are many projects related to it, integrated with demand side management (DSM) or used to set energy prices. All these models are usually used for short periods of time, however, long-term forecasting models are not as common. The main problem is that the longer the forecast period, the greater the uncertainty and the accuracy of the models start being unacceptable. This article tries to find the most accurate and fastest models for long-term demand forecasting and compare the different methods.
One of the main tools used in the energy forecast methods is related to artificial intelligence, because since its conceptualization in 1940 [2], has been an important computational process that helps different sectors. Until this time, different companies, universities and organisations have developed different models in order to predict future energy values.

The first question that arrives is: what is artificial intelligence and how to apply it in the energy sector?
Artificial intelligence is a system that can re-organise or improve its decisions over time depending on the current circumstances. To do it, it is needed to collect and analyse large datasets (Big Data).
The final aim of this system is to be able to find a solution without any human intervention.
The year 1956 is considered the birth of AI, where the event Dartmouth Summer Research project was born[3]. Since this year different models were introduced and scientists realised that many important applications of AI needed a large amount of data that was tough to manage. Due to this fact, AI and Big Data are usually working together. The problem with the first models is that the results were not as good as needed because of the nature of the models used, but a new concept known as backpropagation captures again the attention of researcher groups to use again AI in the 1970s. This concept enhances the way that neural networks improve their accuracy over time, reducing their error. Moreover, the improvement of technology allows the use of heavier models and to use more data due to the simple access and the growth of Big Data and Smart meters.

Even, being AI a good tool for demand forecasting, it is not only related to this. As AI can enable fast and intelligent decision-making, it can be added to different energy problems to solve them, increasing the grid flexibility.
If the grid is separated into the generation, transportation and distribution parts, AI can be integrated into all of them doing different tasks.
In the generation part, can be useful, for instance, for wind and solar forecasting. The management of Big Data can optimise and improve the accuracy of the forecast. The increase of AI in the generation part is pushed by two main innovation trends: decentralization (increase in the deployment of small power generators) and electrification (in transport and buildings mainly).
In the transportation part, AI can help to improve grid stability and reliability. On the other side, in the distribution part, it can optimise the demand-side management.
However, some key points must be solved before its integration such as the availability and quality of the data and the cybersecurity to avoid attacks.

There are some examples of AI integrated into the grid such as EUPHEMIA, which is a coupling algorithm that integrates 25 European day-ahead energy markets. This algorithm tries to join the European electricity forecast markets but also HVAC load management with AI has been used in order to reduce the consumption in buildings[4].

After knowing some possibilities where AI can be integrated, it is possible to classify them, in six main categories[5]:

1. Improved renewable energy generation forecast
2. Maintain grid stability and reliability
3. Improved demand forecast
4. Efficient demand-side management
5. Optimised energy storage operation
6. Optimised market design and operation

The one studied in this report is the improvement of demand forecast, which can enhance the demand-side management in the short-term but also helps political decisions in order to organise the elements needed in the grid.

However, papers done in this research field are not as standardised and are not as easy as other papers to obtain conclusions from it or to compare with different scenarios. Commonly, authors considered neural networks as a black box where their relative features are not really taken into account but it is its knowledge what make the model work properly. Moreover, it is really common to find many papers that use hybrid algorithms in order to obtain better results because the accuracy of the combination is higher than its counterparts.
On the other side, a common mistake is to overfit the models making some noise that can be avoided in order to improve the accuracy but this is difficult to evaluate if it is happening if the raw data is not given.
Therefore, it is difficult to trust other papers or to obtain conclusions from them because the fact of extrapolating their results to other cases can be misleading. Different factors that can affect the results as the quality or size of the data, making the results of the paper very specific for this scenario and for this time. Furthermore, the quality of the papers is not the same even using same tools and, also, some authors exaggerate results or manipulate data in order to obtain better results[2].
One important aspect of this article is that different methods are compared for the same scenarios and with the same raw data, where the parameters used are just adapted for each model, making it more reliable and fair to compare between them.
All the same, neural networks are not the only known tool to predict future values but also statistical methods are very well known and are integrated into different sectors for forecast purposes. The difference between them, among others, is that Artificial Neural Networks (hereafter referred to as ANN) parameters are found by the concept of backpropagation previously explained while the coefficients of statistical methods are optimised trying to obtain the lowest root mean square error or other feature that reduce the error of the model.

Formerly, these forecasting processes were carried on by power companies because of their

knowledge in the field, however, new sectors are taking part in the energy forecast because of their familiarity with the forecast field. Most of the forecasting competitions developed from the 1990s to the 2010s were won by unrelated power companies[2].
Truly, does not exist the best forecast tool for a specific field, but it is how the models are applied and developed what matters. Any statistical tool or ANN used by a professional will have better results than any other model developed by someone without experience.

In this article, the models were divided into two main categories: the statistical methods and the ANN. In the state of arts, the fundamentals of the models are addressed and in the methodology section, the tools and the results are depicted. In order to compare them properly, two scenarios were addressed having different shapes and data sizes.
On the other side, the models were created following a one-step prediction where the model predicts one value, calculates the root mean square error (hereafter referred to as RMSE) and will be added to the historical.
Most of the models were developed just taking into account energy in the historical, i.e. using univariate models, but another parameter has been addressed as an input variable in some models: temperature.
Both have a nonlinear relationship, when the temperature decreases, the load also increases because of heating needs, but when the temperature increases, the load also increases because of cooling needs. That fact can create confusion by leading one to think that nonlinear models will suit better than others, but the reality is that linear models work as well as nonlinear with nonlinear problems[2].
In the end, the results and comparisons are shown.

3 Introduction

3.1 Objective

This report aims to **compare different forecasting methods** in order to determine which is the best solution to **predict the energy demand in a long-term period**. Among the possibilities, the methods can be classified into statistical methods and neural networks.
The best solution will be considered the closest one to reality and the process to score the models must be standardised. In this case, the metric to evaluate the model that leads to the best solution is the **root mean square error (RMSE)**, but also, other characteristics of the models such as the **computational cost** or the time needed to execute the codes must be considered for their comparison.

3.2 Scope of the project

The scope of this project is to compare different prediction methods and to determine the **most accurate for long-term periods** in the case of energy demand, taking into account the computational cost, the time to execute the codes, the error metrics and the data cleaning process among other characteristics of the models.
Although different scenarios were studied, all of them were predicted considering the per unit (pu) system **to compare the results easier**.

4 State of the art

In this report, different methods are scored in order to check how they behave to predict future values of electrical demand.
For a long time, people have tried to find out the evolution of the weather through superstitions, but since Aristotle, the studies and analysis of the weather began to be more accurate. It was Aristotle who wrote the first book of meteorology that tries to understand and analysed the observations collected[6].
Nowadays, it exist different tools that help us to do more accurate predictions, but since the first meteorological book, the models are fed by a data historic. Actually, this is a key point to achieve a good result, because the better organised and understood the history is, the most precise the result will be.

The models introduced in this section aim to predict the next values with a historic dataset. All of them have different natures and are manipulated in different ways. Furthermore, some of them, take into account some features of the model that are skipped by others as the seasonality. These models are really helpful in the stock market to do technical analysis of the products or in the energy market to prepare the electrical network in the short- and long-term period. Being a long-term scenario, the main aim is to prepare the actual grid to afford future problems and size the grid properly.
Moreover, fuels can be managed so it is used when it is really necessary and stored in the clue moments, making an increase in renewable and store non-renewable energy when necessary.
The models created in this report are sized **to predict the future values in a long-term period** separated into two cases: one scenario is 4 days and the second scenario consists of 100 days. Maybe it is not the best model to size, prepare or make political decisions related to the future grid but it is **useful to score the models** which is the real aim of the report. Furthermore, 4 days can be considered as a short-term period but this is done to reduce the stochastic behaviour of this scenario's signal which will be addressed later.

The most advanced prediction models can be classified into **statistical** and **neural network** models. In fact, in this section, they are also divided into these two types.

The **statistical methods** are **algorithms** that are supported by **mathematical equations** and can forecast future values. They work making operations based on statistics and probability.
To improve the accuracy, coefficients such as the trend or seasonality can be added and to handle the models, these **coefficients** must be changed, which can be found by other statistical tools or by a grid search analysis which depending on the computer features, will be faster or slower. However, grid searching guaranteed to obtain the best coefficients in a range of possibilities.
This statistical method needs previous data processing to determine how the dataset behaves and how to adjust the equations in order to predict the values. This process is detailed in section 5.2 of page 29.

On the other side, **neural networks** are based on linear regression models stored in nodes, which try **to act as human brains**. That makes that programs based on them can predict, recognise and solve the problems as a human does. It is part of AI and is formed by nodes that create layers. There is always an input layer, an output layer and one or more hidden layers[7]. Each node is connected to others with different weights.
For these kinds of models, it is possible to do a grid search analysis in order to configure them in

the best way possible, however, this process takes a lot of time with supercomputers, therefore the alternative carried out has been a **trial-error process**.
ANN learns from previous values trying to improve its accuracy until the generated error achieves a feasible value. The dataset in these kinds of cases must be prepared to suit the model properly depending on the ANN used and usually is not as simple as the data treatment of statistical methods, which must be taken into consideration.

The statistical models that will be introduced in this report are the **Exponential Smoothing**, **SARIMA** and **SARIMAX**. On the other side, the ANN developed were the **Convolutional Neural Network** and two different variants of **Recurrent Neural Network**.
Those are just a couple of models among the huge possibilities that exist. However, those are very developed and well documented. Moreover, most of them add some features that adapt better to the history as the seasonality, for instance.

As it was said in the preface, there is not any method that can be considered the best model but it depends on the knowledge of the programmers to have a more accurate result. Nevertheless, figure 1, shows a summary of the root mean square error achieved after doing some forecasting exercises with the same datasets. These results were taken from the book "Predict the Future with MLPs, CNNs and LSTMs in Python" by Jason Brownlee. The root mean square error was the error metric also used to score how far or how close were the results achieved to the reality.

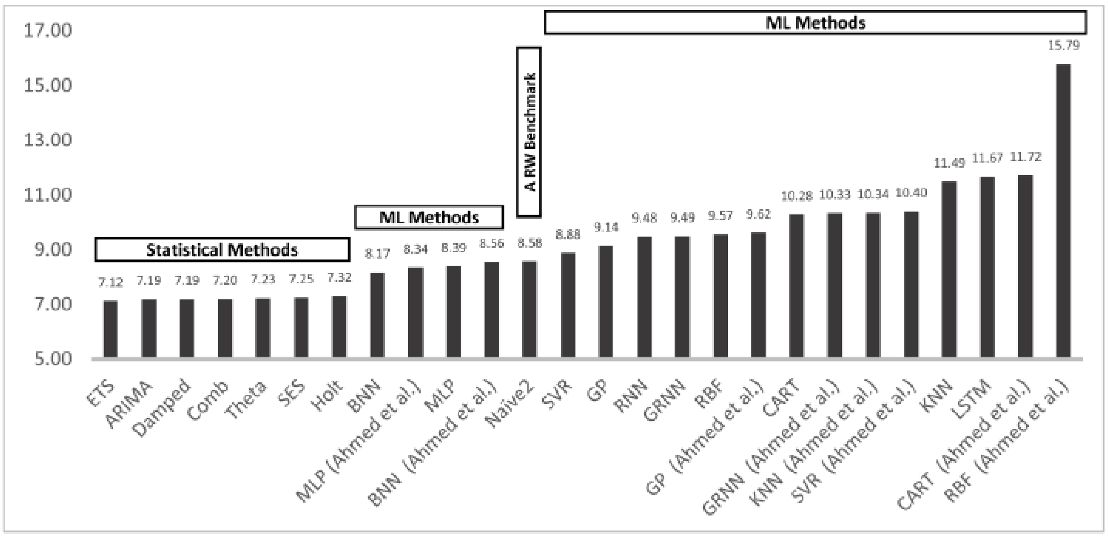


Figure 1: Results taken from Jason Brownlee [1]

With this summary, it can be expected that the statistical methods will give more accurate results in comparison to Artificial Neural Networks. Some authors suggest that this is due to overexertion of the neural networks which is, sometimes, counterproductive.. Obviously, the lower the root mean square error (on top of the graphic results), the better the model. In section 5.5, results of this report and the results of figure 1 will be compared.

In the industrial sector, both types of techniques are really common. For instance, SARIMAX model has been used for electricity price predictions, temperatures forecasting or tourism forecasting among others while ANN has been used, for instance, in eCommerce, finance, healthcare or logistic[8]. That can give an idea of how well adapted can be both types of techniques in today's industry and the range of possibilities.

4.1 Statistical methods

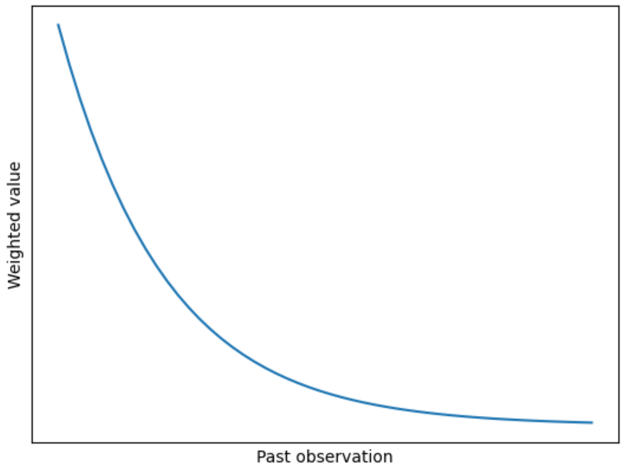
As it was mentioned above, statistical methods do not work learning from errors but thanks to **probabilities**. It interprets how the past behaves under certain conditions and applies the same trend as the past. The size of the dataset can be modified or simplified to reduce the noise and, therefore, it has faster and more accurate results.
The statistical methods here addressed are the **Exponential Smoothing** model, which is a univariate model, the **SARIMA** model, which is also a univariate method, and **SARIMAX**, which can add an exogenous variable that, in this case, will be the temperature.
The Exponential Smoothing method is very simple and easy to use but it does not mean that the results expected will be not very accurate. In fact, this is the prediction method used in Excel to do forecast exercises and as can be seen in figure 1, it shows the best result of this study.
On the other side, SARIMAX and all its variants are very well known. All the coefficient that it has, allows to modify the behaviour of the model to focus on the most important part of the historic, having faster and more accurate models. ARIMA, which is a univariate variant that does not include seasonality on it, is the second-best model of the results of figure 1.

4.1.1 Exponential smoothing model

The exponential smoothing is a **weighted linear predicted model for univariate** data that weighted past observations to predict the future.
The model is composed of a linear sum of recent past observations. The **lags are weighted** with a **geometrically decreasing ration** weight, meaning that more recent lags have a higher associated ratio as can be seen in figure 2. This model can also be known as ETS because of error, trend and seasonality. These are the three key elements of the model[1]. It is a very simple method compared to other possibilities but it allows to be fast for the accuracy obtained.

The exponential smoothing can then be divided into three types: **Single exponential smoothing (SES)**, **double exponential smoothing (DES)** and **triple exponential smoothing (TES)**. The difference between them is the addition of a **trend** behaviour (in DES models) or the addition of **seasonality** (in TES models) to a systematic structure. In this case, the model used this work is the TES model known also as the **Holt-Winter Exponential Smoothing**. In this kind of model is really important to have ordered time series.

Figure 2: Weighted linear sum



Single Exponential Smoothing (SES) It doesn't include any seasonality or trend to the model and it is modelled using the **smoothing factor** or coefficient α that ranges from 0 to 1. An α close to 1 means that the model pay more attention to **most recent observations** called also fast learning models. On the other hand, α close to 0 pays more attention to **far observations** known also as **slow learning models**.
The formula that SES models follow are the one of equation 1.

$$s_t = \alpha x_t + (1-\alpha)s_{t-1} = s_{t-1} + \alpha(x_t - s_{t-1}) \tag{1}$$

$$\begin{cases} \alpha = \text{smoothing factor} \\ s_t = \text{smoothed statistic} \\ x_t = \text{current observation} \\ s_{t-1} = \text{previous smoothed statics} \end{cases}$$

As can be seen in equation 1, the smoothed statistic will be a **weighted average of the current value and the last smoothed statistic**.

Double Exponential Smoothing (DES) The double exponential smoothing **adds support for the trend** in the univariate time series. In this kind of model, the parameter α is used for controlling the smoothing factor and β for **controlling the decay of the influence of the change in the trend**.
The nature of the trend that this model can support is the **additive trend or the multiplicative trend**, depending on whether the trend is **linear** or **exponential** respectively.
The additive trend, one of the most popular, is known also as the Holt's linear trend model.
Nonetheless, there is a problem with this kind of model if a multi-step process is done. In this kind of model, the **trend can change** and not be the same among the steps, therefore, a new parameter that **dampens the trend** φ is added to change the trend over time in this case. **Dampening** means **reducing the size of the trend** over future time steps down to a straight line. This

new parameter controls the rate of dampening that can be, as happened with the trend, **additive** dampening for linear scenarios or **multiplicative** dampening for exponential scenarios.

In this method, if the dampening coefficient is skipped, there are two associated equations.

$$s_t = \alpha x_t + (1-\alpha)(s_{t-1} + b_{t-1}) \tag{2}$$

$$b_t = \beta(s_t - s_{t-1}) + (1-\beta)b_{t-1} \tag{3}$$

Where x_t is the sequence of observations, s_t is the smoothed statics, b_t is the best estimate and the output will be given by an approximation of the smoothed statics and the best estimate of the trend.

Triple exponential smoothing (TES): This last model **adds support for seasonality** and it is also known as Holt-Winter Exponential Smoothing. In case the data shows a trend and seasonal behaviour, the TES model can predict better future values. This model uses the parameters α and β as the previous model but it also adds the γ parameter which controls the influence on the **seasonal component**. This parameter also is set as **additive** or **multiplicative** depending on the linear or exponential process.
This method is the most complete one of all the exponential smoothing models, because it can change the level, trend and seasonality patterns. If a seasonal parameter is added, it must be add also the number of time-steps in a seasonal period.
The equation that the TES model follows is represented in equation 4.

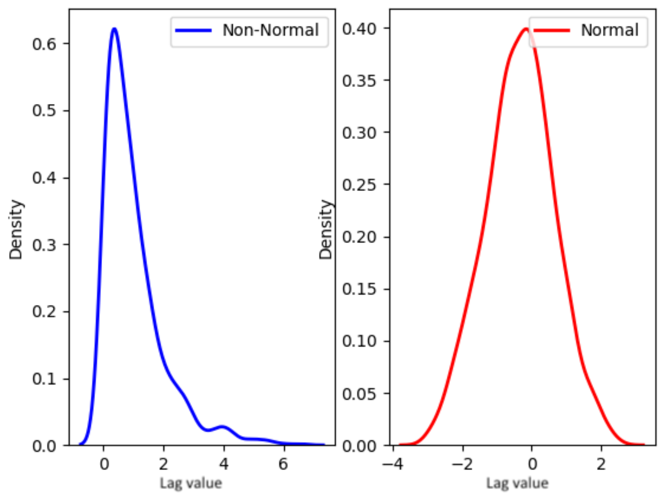
$$\begin{aligned} s_t &= \alpha \frac{x_t}{c_{t-L}} + (1-\alpha)(s_{t-1} + b_{t-1}) \quad \text{Overall formula} \\ b_t &= \beta(s_t - s_{t-1}) + (1-\beta)b_{t-1} \quad \text{Trend smoothing} \\ c_t &= \gamma \frac{x_t}{s_t} + (1-\gamma)c_{t-L} \quad \text{Seasonal smoothing} \end{aligned} \tag{4}$$

In this case, the formula that was added is c_t equation which is related to the seasonal statics. c_t will be considered as a sequence of seasonal correction factors and L is the length of the seasonal change.

Moreover, this model has in the python library a parameter called Box-Cox which transforms the data to a normal distribution to improve the predictive power. It does this because of cutting away the white noise thanks to normalizing the errors[9].

```
from statsmodels.tsa.holtwinters import ExponentialSmoothing
```

Figure 3: Box-Cox transformation



4.1.2 SARIMA model

SARIMA model is a statistical model which tries to forecast future values by taking into account some past parameters and, maybe, modifying them. For this approach different methods are integrated in this model which is a combination of an **autoregressive model** (AR), a **moving average model** (MA), an **integration** (I) and a **seasonal** component (S).

The forecast can be done with stationary time series that can show a trend in future values. That means that knowing the stationary condition of the time series is important, to know how to proceed with the forecast. To realize if a time series is stationary or not, different methods can be applied, for instance, it is possible to use the **Augmented Dickey-Fuller** method that uses two possible hypotheses: the time series is **not stationary (H_0)** or the time series is **stationary (H_A)**. Fortunately, Python has libraries that can do the stationary test giving the results of the stationary condition. The library used for this is called statsmodels.tsa.stattools which integrates an adfuller tool.

```
from statsmodels.tsa.stattools import adfuller
```

This tool gives back a parameter called p-value which can estimate the stationary condition of the time series. If it is lower than 0,05, can be consider as stationary (H_A)[10]. If the time series is considered non-stationary (H_0), then **integration** of the time series must be developed in order to become stationary the data.

On the other side, the configuration arrangement will be set after a grid search. This tool is not as selective as the Augmented Dickey-Fuller test, the autoregressive function or the partial autoregressive functions but thanks to the actual technology development is easy to integrate and reduce the non-intuitive possibilities and the errors of the model.

The **autoregressive model (AR)** tries to measure the **impact of last values** in the data frame in next predicted value. The AR model follows equation 5.

$$X_t = C + \phi_1 X_{t-1} + \epsilon_t \tag{5}$$

In this equation, X_t is the **predicted value**, i.e. the value we want to know taking into account past values. C is a **constant factor** and X_{t-1} is the **value of the last unit of time** which is weighted by ϕ, which is a numeric **constant** that shows how much this value affects the present one. ϕ ranges from -1 to 1. Finally, ϵ_t is the **residual** value that can be considered as the difference between our prediction for period t and the correct value. This value is an uncertainty until we know the real value.
The key point of autoregressive model is to know how many lagged values must be taken into account in our model. Note that equation 5 shows the impact of the previous unit time value, which can be depicted as AR(1). Therefore, the number of lagged values is displayed in AR(**p**) The more lagged values we include in the model, the more accurate it will be but also the more complex. That means, that more likely some lagged value will be not significant[11]. Then, knowing how many and which lagged values we can include in the model is a crucial in this case.

To achieve this, two correlation models can be studied so we can deduce the correlation between present values and past values. These two tools are the **autocorrelation function** (ACF) and the **partial autocorrelation function** (PACF). Both aims to study the influence of the past in the present values. The ACF shows the correlation between two values of the time series $(corr(y_t, y_{t-1}))$.
On the other hand, the PACF finds a correlation of the residuals with the next lag value[12].
If ACF values are inside the blue region (in section 5.3.3, the charts with the blue region can be seen), it means that there are not statistically significant[13].
The most useful for identifying the order of an autoregressive model is the PACF but the most useful for the moving average, which will be addressed later, is the ACF[14].

Moving average (MA) uses also past values as the AR model, but instead of using the values themselves, it uses the **difference between the estimations and the actual values**, i.e., it is considering past residuals. The MA(1) model follows equation 6 and, as happened with the AR model, it uses a coefficient that shows the number of lagged values that are considered on it (MA(**q**)).

$$r_t = C + \theta\epsilon_{t-1} + \epsilon_t \tag{6}$$

The equation is really similar to the AR model equation. In fact, as happened before, C keeps for a **constant** factor and ϵ_t represents the **residual** for the current period. The difference between the equations is that, this time, the lagged value is not chosen to predict the future but ϵ_{t-1} which is a **residual for the previous period**. Moreover, θ indicates the impact that past residuals have on the present one. It is common to represent with a different letter than the AR model to don't get confused.
In this model, to estimate the ϵ_t it starts from the beginning to predict each value and obtain a residual value which is the difference between the estimation and the real one[15].
For this model, contrary to AR model, the best correlation function that shows the number of residuals that must be taken into account is the ACF.
In the final model, the p and q can but doesn't need to match[16].

Until now, a model that uses past values as a benchmark and error terms of last periods is created.

In this model, an **integration order** (d) can be added in order to **ensure a stationary condition** which can give the forecast estimation. This order indicates the number of times that the model needs to be integrated to become stationary. For instance, an ARIMA(1,1,1), where ARIMA(p,d,q), is depicted in equation 7

$$\Delta P_t = C + \phi_1 \Delta P_{t-1} + \theta \epsilon_{t-1} + \epsilon_t \tag{7}$$

In the equation 7, $\Delta P_t = P_{t-1} - P_t$. That means, that ARIMA models are just ARMA models that integrate a new time series that guarantees a stationary condition.[17] It is important to take into account that dealing with integrated values means losing d-many observation because the first value must be erased since there is no previous one to make the difference.

$$\Delta P_1 = P_0 - P_1 \rightarrow \text{where } P_0 \text{ doesn't exist}$$

Moreover, the time series changes from being values to the difference of it.

In addition, there is a **seasonal parameter** that can speed up and improve the accuracy of the models. It needs to specify the parameters related to the trend (p, d, q) and related to the seasonal parameters (P, D, Q, m). The trend elements were explained above and the seasonal elements are:

- P: It is the seasonal regressive order.
- D: It is the seasonal difference order.
- Q: It is the seasonal moving average order.
- m: It is the number of time steps for a seasonal period.

The final SARIMA model can be expressed like SARIMA(p,d,q)(P,D,Q)m. It is obvious that the m parameter influences the seasonal parameters P, D and Q. If P is 1, it means that only the last season period will be taken into account but if the P is 2, the two previous seasonal periods are included in the model.
As happened with the moving average or the autoregressive parameters, checking the ACF or the PACF can be interesting to guess if the time series add a seasonal component to it. If lags values are out of the band taking with a repeated distance between them, it means that a seasoned behaviour is integrated into the model.
The difference in the SARIMA model can be found in the lag period terms because there are multiplying the m factor to skip some terms of the model that are not as important. Therefore, the final equation will be the equation 8.

$$\Delta P_t = C + \phi_1 \Delta P_{t-1\cdot m} + \theta \epsilon_{t-1\cdot m} + \epsilon_t \tag{8}$$

In equation 8 can be seen a subindex m which specifies the lag distance between the previous parameter that the model must respect in order to set a seasonality.

4.1.3 SARIMAX model

SARIMAX model is a step ahead of last model because it includes an additional value for an **exogenous data** that can affect the final model, improving the accuracy of it. An exogenous variable is a variable that is **not affected by others** and on which the final data depends[18]. The equation of SARIMAX is similar to SARIMA's equation but it includes a new coefficient to weight the exogenous variable[19]. To clarify, the final equation model without seasonality will be as follow:

$$\Delta P_t = C + \beta X + \phi_1 \Delta P_{t-1 \cdot m} + \theta \epsilon_{t-1 \cdot m} + \epsilon_t \tag{9}$$

The equation 9 is the SARIMAX equation for a case where the p, d and q are neglected and the P and Q parameters depend on m[20].
If the equation is broken down, the unique new parameters are βX, which are related to the exogenous variable. X is the exogenous variable itself and β is the weighted parameter for the exogenous variable.
In these two scenarios, the exogenous variable will be the **temperature** in London or in Canarias, which were collected thanks to the Power Nasa tool. That is better explained in section 5.1, page 27.

4.2 Neural Networks

Neural Network is part of the Machine Learning technology that tries **to behave as a human** does to recognize a pattern and solve problems with computer programs. As machine learning is **part of AI**, the aim is to make computers interact mimicking what a human will do. Throughout history, there have been numerous examples of machines beating humans in games as happened with Garry Kasparov, world chess champion, when he lost playing chess against the AI Deep Blue in 1997.

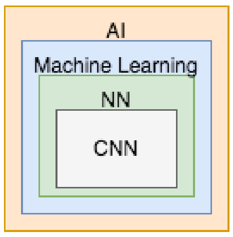


Figure 4: Nested concepts

The neural networks are composed of an input layer, one or more hidden layers and an output layer. Neural Networks use this name because the idea is to create a similar network to the **brain's neural system**. Each neuron is mimicked by a **node** in the NN, which is a simple linear regression model composed of input data, weights that determine the importance of each variable, bias or also known as threshold and an output[21].
The formula of a node is represented in equation 10.

$$\sum_{i=1}^{m} w_i x_i + \text{bias} = w_1 x_1 + ... + w_n x_n + \text{bias} \tag{10}$$

After checking if the output of each node is above the threshold of each one, the node is activated giving the information to the next layer. In case it is not above, data is not passing to the next layer as can be seen in equation 11.

$$\text{output } f(x) \quad \begin{cases} 1 \text{ if } \sum w_1 x_1 + b \geq 0 \\ 0 \text{ if } \sum w_1 x_1 + b < 0 \end{cases} \tag{11}$$

In these equations, the input values will be multiplied by their weights giving different results that after summing all of them, it will be introduced in an activation function that will give an output. If this output, exceeds the threshold it activates the function and behaves as the input of the following node.

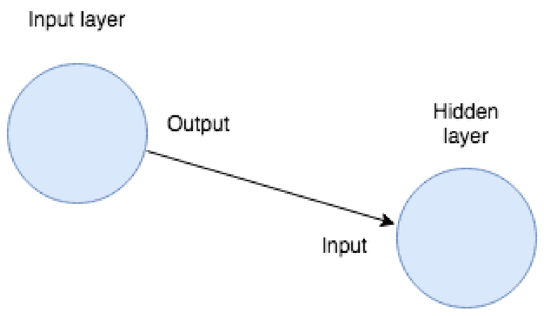


Figure 5: Neural nodes information transmission

Neural networks work by training data to learn how to predict future values improving their accuracy. The main category classification that NN has are the ones used **to classify** and the ones used **to cluster**.

4.2.1 Convolutional Neural Networks (CNN) model

This kind of NN was created to recognise two-dimensional images. Nowadays is used for image recognition, pattern recognition and for computer vision[22].
It can be also used in time series prediction models and, actually, many types of CNN models can be used for each specific type of time series forecasting problem.

This kind of model requires to have a temporal ordering input data to learn from lags observation series.
Moreover, to model this NN, data preparation is needed, as explained in page 29 section 5.2, because the input data can not be the same as the input time series data for a statistical model but as the following matrix.

$$\begin{array}{cc}
input & output \\
[1,2,3] & [4] \\
[2,3,4] & [5] \\
[3,4,5] & [6] \\
[4,5,6] & [7] \\
[5,6,7] & [8] \\
[6,7,8] & [9] \\
[7,8,9] & [10]
\end{array}$$

Convolutional networks are composed mainly of three types of layers: **convolutional layer**, **pooling layer** and a **fully-connected layer**.
The convolutional layer must be the first layer in the network but it can also be found in hidden layers. The fully-connected layer must be the final layer, being the pooling layer the middle layer if there are any.

Convolutional layer: In this layer happens the **majority of the computational work**. It is made up of input data, a filter and a feature map.
The input layer is a three-dimensional matrix that a feature detector, also known as a **kernel** or **filter**, will go across to obtain values. The feature detector is a two-dimension array that **put some weights** on the input layers number in order to convert data into numerical values, allowing to extract relevant patterns

Pooling layer: It allows to **reduce** the number of parameters in the input and, therefore, it is **losing some information** in the model. However, it also reduces complexity, improving efficiency. This layer is also composed of a **filter** with the difference that the filter doesn't have any weight but an **aggregation function**.

Fully-connected layer: The aim of the layer is **to connect** each node of the output layer directly to a node in a previous layer.

The way of creating a CNN with Python can be, for instance, as the following code shows which uses a convolutional layer followed by a pooling layer, then a flatten layer that reduce the feature maps to a single one-dimensional vector and a dense layer. The functions use a rectified linear unit and the mean square error to optimize the model.

```
# the model
model = Sequential()
model.add(Conv1D(64, 2, activation='relu', input_shape=(n_steps, n_features)))
model.add(MaxPooling1D())
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

4.2.2 Recurrent Neural Network (RNN) model

The other NN that can be seen in this report is Recurrent Neural Networks, which are used to solve common temporal problems. This can be useful in language translation, speech recognition or image captioning. It can use sequential data or time series data, which are the interested ones in this report. As happened with the others NN, it uses training data to learn and improve the model, but in this case, they are different because of their "memory" that manages the information to distinguish between relevant information and useless ones. That can be done, using past information of the last forecasts, i.e. using **recurrent information as feedback** to improve the model.

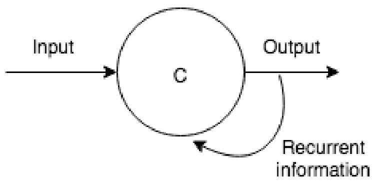


Figure 6: RNN cell

In convolutional neural network, each cell uses a different weight, however, in this case, RNN **layers share some parameters** like the weight. That means that there is only one weight per layer.
On the other hand, to determine the gradients, RNN uses a **backpropagation through time (BPTT)** algorithm, which is used in sequence data. In this case, the model trains itself by calculating errors from the output to the input layer and the calculations will improve the accuracy of the model. The difference from classic backpropagation is that **BPTT sum the errors at each time step** because in this case, each layer of the model share parameters, unlike in feedforward network[23].
In addition, unlike CNN, the inputs and outputs don't need to match but it can use different arrangements depending on the application.

As happen in CNN, the **common activation functions** of the cells are **sigmoid**, **hyperbolic tangent** and **ReLu**.

However, this kind of NN has two problems which are called the **exploding gradient** and the **vanishing gradient**. If the gradient is too small, it will continue to become smaller until it becomes insignificant, which is known as the vanishing gradient. On the other hand, if the exploding gradient happens, the gradient will be so large that the model will become unstable.
That means, that adding more information will feed better the model but it will add more difficulties to running it. The simplest solution to avoid these problems is **to reduce the number of hidden layers**, making the model, probably, less accurate. In this kind of NN, long time lags are inaccessible because the backpropagated error blows up or decays exponentially[24].
Over time, has emerged different variants of RNN in order to overcome the exploding and vanishing gradient problems such as the Bidirectional recurrent neural networks (BRNN), the Long short-term memory (LSTM) or the Gated recurrent units (GRUs).

BRNN are bidirectional RNNs that uses future data to improve the predictions. This NN is useful in translation applications or voice recognition for instance, but in energy forecast scenarios maybe is not the best tool.
LSTM is the most popular one which gives a **solution for the vanishing gradient** problem. In this variant of RNN, if the past prediction which affects the new variable is not a recent value, it manages to use it anyway unlike classic RNN. That happens due to three gates integrated into the hidden layer cells called the input, the output and the forget gate which is an analogy of **write**, **read** and **reset**[25].
Finally, the **GRUs** solve the short-term memory problem of RNN. Unlike the LSTMs that use states in the hidden layer, tha GRUs uses hidden states with two gates, the reset and the update gate.

Long short-term memory (LSTM): It is the most popular RNN introduced by Sepp Hochreiter and Jurgen Scmidhuber that solves the vanishing gradient problem, i.e. that can forecast the output **using not recent past observations**. This system must **store information** for an arbitrary duration, be **resistant to noise** and be a **trainable** system.
In addition, the system will **solve the vanishing** and the **exploding gradient** problem thanks to the internal structure of the unit. This structure works using two gate units out of three to **open** and **close** the error access within each memory[26].
As was said before, the LSTM neural network uses **three gates** that manage the states of the nodes in order to decide which are the relevant information that the system must save. That

three gates are an analogy to the write, read and reset tools that in this case choose **what to forget** (forget gate), **what to add** or update (input gate) and **which stored information shall be used** at each moment[27]. All this is managed by setting the gates in a range of 0 to 1, meaning 1 that the information is absolutely useless, it must be updated or added completely or the information must be read.
There are different types of LSTMs configurations, for instance some of them are:

- Vanilla LSTM
- Bidirectional LSTM
- CNN-LSTM

Although Vanilla LSTM gives really good results and is very simple, in this report the **CNN-LSTM** arrangement will be carried on because LSTMs require stationary data and for that, the usual method is to trim the dataset. Even being a good solution, the purpose is to predict future value with past values without trimming it to have the largest history possible.
However, the CNN-LSTM neural network works with the CNN model to interpret subsequences of input and will provide the output to the LSTM model to interpret it.
This NN was designed for sequence prediction with spacial inputs such as images or videos[28] and it is really useful for visual time series problems.
The basics of the NN are to use the **CNN layer for feature extraction** on input data and to use the **LSTM to do the forecast** by interpreting the features. There is a single CNN model and a sequence of LSTM models (each time step has one)[28].

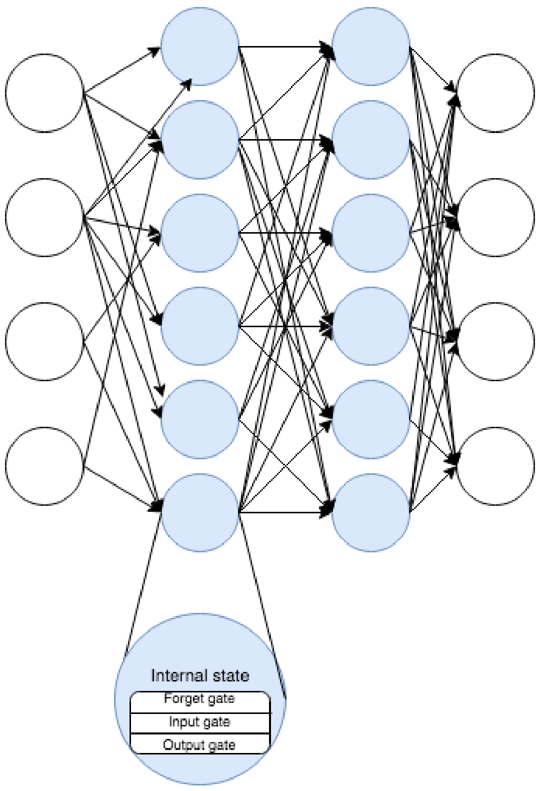


Figure 7: LSTM cell architecture

5 Methodology

In this section, the different methods explained in section 4 will be addressed. They share some of the functions, like the grid search, which is a process where the best combination of parameters of the models is searched among those given by the user. All the statistical methods, due to the easier computational cost in comparison to NN, use a grid search function in order to find the most accurate one. However, in the case of NN is also possible but its computational cost is higher due to its complexity and can means several hours or days running the codes[1].
Two different scenarios will be studied in order to compare both of them and to know how the parameters related with the model changes depending on the situation. All the models work in p.u. values in order to make easier the comparison. The first one doesn't have any seasonality, making more difficult to predict and in addition as it is not an accumulation of different users energy consumption, but only one, the shape is more arbitrary. Therefore, a second scenario is added where the data is an accumulation of users energy consumption and it has seasonality.

All the forecast methods work with **one-step prediction**s, i.e. predict one value and check the actual value to calculate the RMSE instead of using multi-steps predictions. The best method should be a multistep forecast because the aim is to analyse how the grid will behave in the future with long scope, however, in order to analyse the different methods, can be important to check how the feedback of the parameters works. The results will be better than multistep forecasting results which must be taken into account.

5.1 Data to analyse

The first scenario studied in this work is the forecast of a dataset that collect 5.567 smart meters house consumption. Only the data of one smart meter will be used. The data is collected through different smart meters address per household, which are in **London**, United Kingdom. The data collection ranges from October, 12th 2012 to February, 27th 2014. It is gather each half an hour and, instead of using the sum of all the counters to simplify the prediction, as a first step, just one counter will be studied making the data more arbitrary and random. In the second scenario, this will be reduced thanks to the nature of the dataset.
On the other hand, due to the data is given each half an hour, the first step is to convert in an hourly intervals so it can also suit the second variable added in this scenario which is the temperature.
Temperature was taken from NASA's dataset[29] that gives temperatures in hourly intervals. As can be seen in figure 8, it is really difficult to predict which will be the following value as the signal doesn't follow any pattern. There are two spikes close to 5 kWh but with different shapes. These two spikes happened just two times in 1 year and 4 moths of the data collection period and both of them occurred in February, i.e. the winter season. It makes sense that the increase of these values happened due to the conditioning increase. The second parameter added in the dataset is the temperature. This variable shows properly the winter and summer seasons. However, if the graph maybe shows a strong **correlation** among them, the real correlation is just 0,105.

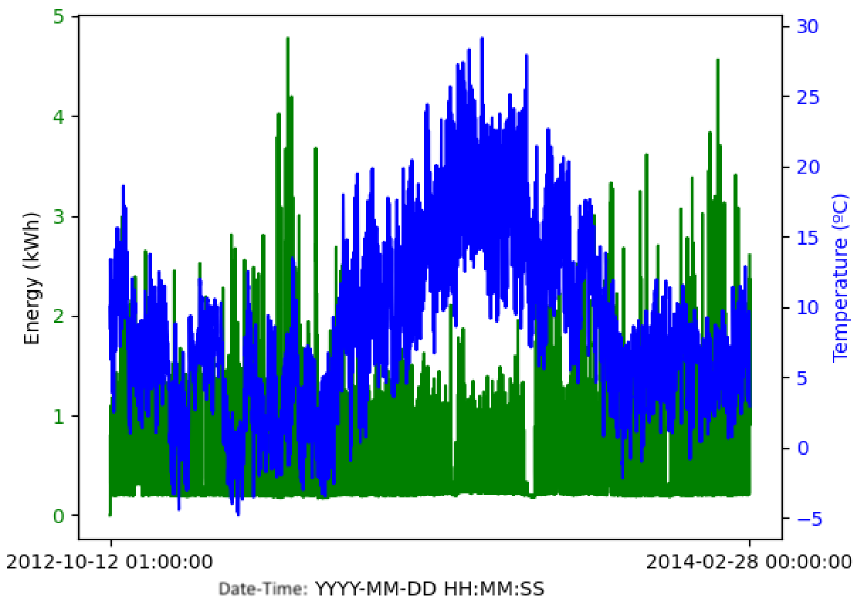


Figure 8: London shape

The second scenario takes the total energy consumption of **Canary Islands** from January, 2nd 2015 to January, 2nd 2020. That makes 5 years of energy consumption data. The idea of this time interval is to avoid the exceptional energy data from the pandemic period and therefore it finish at the beginning of 2020. With this dataset and the following forecast methods, the energy without pandemic can also be predicted.
The dataset uses daily energy instead of hourly and also the temperature taken from NASA data resources[29] is composed by daily intervals.
In the second scenario, as there is an accumulation of data, the prediction is easier. Moreover, there is a seasonality added in the scenario that simplify even more the forecast, but later will be shown that the seasonality will be skipped sometimes in the tools used.
The data this time shows a seasonality on it in both variables, being them phase out 3 month approximately during the winter and adjusted in the summer season. In addition, the graph shows a stronger correlation between the variables in comparison to the last scenario and, in fact, the correlation between them is 0,43.
Unlike previous scenario, in Canarias, the largest energy consumptions can be found in the summer season and the lowest values during the winter season. That can be due to the increase of the population during summer period because of tourism.

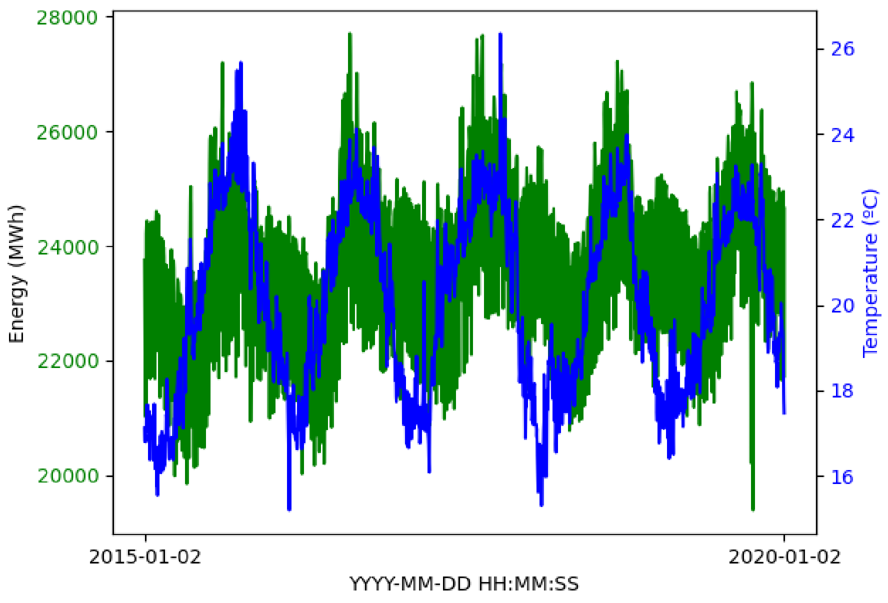


Figure 9: Canarias shape

5.2 Data treatment

There are two main methods of data processing **depending on the method** that we are using. The treatment methods depend on the method used, i.e. statistical methods or neural networks. In both cases, it is better to assure that there is no missing value in the set. This step is one of the most time consuming parts, but it is needed to pay attention to this part in order to easily obtain more accurate results. The input data of the models are completely different. Statistical models organised the values with a time dependency. However, ANNs are fed by samples that contain a set of values.
The step followed to organise the inputs are detailed bellow.

5.2.1 Cleaning data

In this section, the steps followed **to order the information and join** in one data frame will be shown. In order not to be redundant, only the case of London will be explained, showing at the end only the result of the Canary Islands so as not to repeat the same process.

Two different datasets will be ordered and merged to use as the input data of future models. The first dataset is the energy dataset which looks as follows:

```
          LCLid                           tstp  energy(kWh/hh)
0     MAC000002  2012-10-12 00:30:00.0000000               0
1     MAC000002  2012-10-12 01:00:00.0000000               0
2     MAC000002  2012-10-12 01:30:00.0000000               0
3     MAC000002  2012-10-12 02:00:00.0000000               0
4     MAC000002  2012-10-12 02:30:00.0000000               0
```

```
...            ...                            ...              ...
1222665  MAC005492  2014-02-27 22:00:00.0000000          0.182
1222666  MAC005492  2014-02-27 22:30:00.0000000          0.122
1222667  MAC005492  2014-02-27 23:00:00.0000000           0.14
1222668  MAC005492  2014-02-27 23:30:00.0000000          0.192
1222669  MAC005492  2014-02-28 00:00:00.0000000          0.088
```

It has three columns joined to an index column. The first column is related to the smart meter name, which identified which consumption point is responsible for each consumption data. The second column depicts the time when the data was collected and the last column means the energy consumed in the last 30 minutes. There are in total 1222670 values. In this series, there is one missing value for each smart meter in the range from 15:13 to 15:25 on December 18, 2012.
To reduce the dataset, only one counter will be used. After choosing the MAC000002 counter, the dataset will be reduce as it is shown.

```
           LCLid                         tstp  energy(kWh/hh)
0      MAC000002  2012-10-12 00:30:00.0000000               0
1      MAC000002  2012-10-12 01:00:00.0000000               0
2      MAC000002  2012-10-12 01:30:00.0000000               0
3      MAC000002  2012-10-12 02:00:00.0000000               0
4      MAC000002  2012-10-12 02:30:00.0000000               0
...          ...                          ...             ...
24136  MAC000002  2014-02-27 22:00:00.0000000           0.416
24137  MAC000002  2014-02-27 22:30:00.0000000            1.35
24138  MAC000002  2014-02-27 23:00:00.0000000           1.247
24139  MAC000002  2014-02-27 23:30:00.0000000       1.2180001
24140  MAC000002  2014-02-28 00:00:00.0000000           1.387
```

On the other side, the **weather dataset** is formed by the time where the data was collected (year, month, day and hour columns), the temperature and the precipitation. There are 12120 values, which are less than the last dataset. That is because the energy values are collected each half an hour but the weather values are collected hourly. This must be fixed in order to merge the values.

```
       YEAR  MO  DY  HR    T2M  PRECTOTCORR
0      2012  10  12   0  11.08         0.04
1      2012  10  12   1  10.01         0.04
2      2012  10  12   2   9.64         0.03
3      2012  10  12   3   9.49         0.02
4      2012  10  12   4   9.09         0.02
...     ...  ..  ..  ..    ...          ...
12115  2014   2  28  19   4.06         0.22
12116  2014   2  28  20   3.50         0.08
12117  2014   2  28  21   2.92         0.03
12118  2014   2  28  22   2.55         0.02
12119  2014   2  28  23   2.34         0.01
```

First step is to join the weather time variables, i.e. the year, the month, the day and the hour, convert to a **date-time** format and **set as the index**. That can be done by the next code.

```
dfweather.columns=['year', 'month', 'day', 'hour', 'temperature (C)',
        'precipitation']
dfweathertime=pd.DataFrame(pd.concat([dfweather['year'], dfweather['month'],
        dfweather['day'], dfweather['hour']],axis=1))
dfweathertime=pd.to_datetime(dfweathertime)
dfweathertime.name='time'
dfweather=pd.merge(dfweather, dfweathertime, how='outer', left_index=True,
        right_index=True)
dfweather=dfweather.set_index(['time'])
```

In the energy dataset, the first step is to convert energy values to numeric values, considering all missing values as NaN. After, because of the not matching range of values, the half an hour data will be converted as hourly data, making it possible to combine with the weather dataset. Beyond these steps, the data will be handled as the weather dataset, converting time to a date-time format and setting the time as the index. It is done using the following code.

```
mac['energy(kWh/hh)']=pd.to_numeric(mac['energy(kWh/hh)'], errors='coerce')
first_value=mac.iloc[0]['energy(kWh/hh)']
variable=[]
index=[]
for i in range(0,len(mac)):
    variable.append(mac.iloc[i]['energy(kWh/hh)']+
                mac.iloc[i-1]['energy(kWh/hh)'])
variable[0]=first_value
index=mac['tstp']
serie_index=pd.Series(index)
serie_index = serie_index.to_frame(name='index')
serie_variable=pd.Series(variable)
serie_variable = serie_variable.to_frame(name='energy (kWh)')
dfenergy=pd.concat([serie_index, serie_variable], axis=1)
dfenergy=pd.DataFrame(dfenergy)
dfenergy['index']=pd.to_datetime(dfenergy['index'])
dfenergy=dfenergy.set_index(dfenergy['index'])
```

Finally, last step is to combine both data frames and save as .csv file, having one input dataset that collects all the ordered information in one file. That can be done with the following code.

```
dftotal=pd.merge(dfenergy['energy (kWh)'], dfweather['temperature (C)'],
        how='outer', right_index=True, left_index=True)
dftotal=dftotal.dropna()
dftotal.to_csv("final_dataframe.csv")
```

And the information saved will be the next one:

```
                     energy (kWh)  temperature (C)
2012-10-12 01:00:00         0.000            10.01
2012-10-12 02:00:00         0.000             9.64
2012-10-12 03:00:00         0.000             9.49
2012-10-12 04:00:00         0.000             9.09
2012-10-12 05:00:00         0.000             8.80
...                           ...              ...
2014-02-27 20:00:00         2.372             3.93
2014-02-27 21:00:00         1.480             3.50
2014-02-27 22:00:00         0.899             3.15
2014-02-27 23:00:00         2.597             3.05
2014-02-28 00:00:00         2.605             2.95
```

The final dataframe is formed by 12070 values in an hourly range starting on October 12th, 2012 and finishing on February 27th, 2014.

Same must be done with the second scenario (Canarias scenario). To sum up, just the result will be shown.

```
       Unnamed: 0  energy (kWh)  temperature (C)
0      2015-01-02     23753.868            17.12
1      2015-01-03     23016.758            16.80
2      2015-01-04     22156.590            16.81
3      2015-01-05     23511.971            16.94
4      2015-01-06     21020.222            16.79
...           ...           ...              ...
1822   2019-12-29     23074.565            19.39
1823   2019-12-30     24879.108            19.22
1824   2019-12-31     24948.787            18.44
1825   2020-01-01     21702.125            17.79
1826   2020-01-02     24655.193            17.46
```

The value range is from January 2, 2015 to January 2, 2020 and the information is collected every hour.

5.2.2 Data treatment for neural networks

Neural networks inputs are not similar to statistical methods inputs. The neural networks **learn from previous values** without paying attention to the time. That means that no missing value must be founded and a **periodicity must be followed**.
On the other hand, neural networks can manage the input data without as many previous steps as happen with other models. In fact, one simple transformation is needed to fit the model to a CNN or LSTMs NN.
This step is **to transform a two-dimensional structure** into a **three-dimensional structure**.
The way of doing this is **to develop a supervised series** from a time series and **reshape** the input matrix into a three-dimensional one.
Moreover, there is a data split in order to differentiate between the learn data and the test data, as happen with the statistical methods. However, in NN the model will learn from previous values constantly not as the statistical methods that choose the best configuration method after studying one instance.
Therefore, the first step done in the NN analysis studied in this report is the train-split function.

Train-Test Split: As neural networks learn from previous values, the objective of this step is to separate the values into train and test set, where the train set will feed the neural network changing their parameters to improve its accuracy and the test set is done to score the model after doing the prediction.

After this step, the time series must be converted so it suits the CNN or the LSTMs models with the three-dimensional input as it was said before. Then, next step is to convert, the time series as a supervised time series.

Time series supervised: The aim here is **to set the previous values** that the NN must follow to guess the future value in each observation.
Time series are values that depends on time and, for statistical methods input, can be read as follow:

time	*value*
01/01/200	1
02/01/200	2
03/01/200	3
04/01/200	4
05/01/200	5
06/01/200	6
07/01/200	7
08/01/200	8
09/01/200	9
10/01/200	10

However, the way of introducing the values in a NN in a univariate scenario, must be as follow:

input	*output*
$[1, 2, 3]$	$[4]$
$[2, 3, 4]$	$[5]$
$[3, 4, 5]$	$[6]$
$[4, 5, 6]$	$[7]$
$[5, 6, 7]$	$[8]$
$[6, 7, 8]$	$[9]$
$[7, 8, 9]$	$[10]$

The matrix is time independent and there is some lag values that it is possible to extend as much as possible. Each column is a **feature** of the dataset and each row is a **sample**.
Nevertheless, it is a two-dimension matrix and as it was said before, a three-dimensional input is required, meaning that another category is missed.

3D data preparation: The aim of this step is to convert a two-dimensional structure into a **three-dimensional** one.
The two dimensions that the matrix has until now are the:

- Sample: one sequence of values
- Feature: an observation of a certain time

The missing dimension is the **time step**, which is **one point of observation in the sample**. To summarize, a sample is composed of multiple time steps and a time step is composed of one or multiple features.
This change in the dimensions can be done easily with the next tool:

```
X = X.reshape((samples, time_steps, feature))
```

The most common feature in the cases cover by this report is 1.
On the other side, the sample and the time steps values can be known from the previous shape

of the matrix, i.e. after setting it as a supervised series. Knowing that, a 3D data preparation function can be prepared for multiple possibilities as can be seen in the next code:

```
X = X.reshape((X.shape[0], X.shape[1], 1))
```

Being X a matrix of a supervised series.

After this, the way of creating a NN with Python can be as easy as writing the following code:

```
model.add(LSTM(32, input_shape))
```

where 32 is the number of the units refers to the first hidden layers.

5.3 Algorithms

All the algorithms are transformed to per unit values to properly compare the different models, making the final prediction with the real values. On the other side, different computers were used to speed up some processes. This fact must be considered in order to analyse the results. In this report, the computers used had the following features. The first one is made up of a 2,4 GHz unique CPU and 8 GB of RAM. The supercomputer of the UPC department uses 32 GB of RAM and the last computer used is a PC of 2,60 GHz and 16 GB of RAM. The naive model and the exponential smoothing were executed with the 8 GB RAM computer, SARIMA and SARIMAX were developed by the supercomputer and the ANNs were created with the 16 GB RAM PC.

5.3.1 Naive model

Using a naive model can be the **foundation** of this study as it can bring an idea to know which models work better compared with this simple model. To do it, a **mean**, **median** and a **persist** model will be the tools that will compose the forecast model, being the user, the one who chooses the method used.
The **mean** model makes an average of the n last observations of the dataset. That means that in a dataset like the following one:

```
[1, 2, 4, 5, 5, 6, 6, 7, 8, 10]
```

the forecast value that will continue 10 will be $5,4$ if 10 last values of the dataset were taken into account and 10 if just the last value was taken into account. That creates a dependence in the model not only on the methods but also on the n last values used.
On the other hand, if the 10 last values of the dataset were taken into account using the **median** instead of the mean, the result will be $5,5$. Finally, an additional input, called offset, that tries to take into account the seasonal behaviour of the dataset will be introduced. This variable tries to make a prediction by discounting the values outside the season. That can weigh more on some past observation rather than the others depending on the value set. For instance, if the dataset introduced is the following one:

```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

If an offset of 3 is set, a season of the step periods will be understood, which means that all the methods applied will be done given a space of three time steps. That is that a mean value of $n = 1$ is 1, the mean value of $n = 2$ is 1 and the mean value of $n = 3$ is also 1, because is discounting the space of three time interval. This simple forecast can be carried on thanks to the following code:

```
def simple_forecast(history, configuration):
  #the configuration is the n last values used, the offset and the type
  n, offset, prediction_method = configuration
  # if persist value, ignore other configuration
  if prediction_method == 'persist':
        return history[-n] #use last values
  # collect values to average
  values = list()
  #organise the data depending on the offset component
  if offset == 1:
        values = history[-n:]
  else:
        # skip bad configurations
        if n*offset > len(history):
          raise Exception('The configurationuration is beyond end
                of data: %d * %d > len(data). It is not posible to do
                the forecast' % (n,offset))
        # try and collect n values using offset
        for j in range(1, n+1):
          m = j * offset
          values.append(history[-m])
  # check if we can average
  if len(values) < 2:
        raise Exception('Imposible to calculate average')
  # mean of last n values
  if prediction_method == 'mean':
        return np.mean(values)
  # median of last n values
  return np.median(values) #Is better for non-Gaussian distribution
```

This code can be used to forecast the next time-step value, however, to know the best configuration that can be used with this method, extra functions are needed. The rest of the code will try to show the best configuration to the user in order to use it in the forecast for a given dataset. To know which configuration is the best, an iterative method must be used where the root means square error will be the key element used to score the method.

Therefore, some arrangement must be developed in order to calculate the errors and the result of the method. First, two functions, one for calculating the root mean square error and one to split the dataset in a train and test datasets will be developed. As it is said above, the root mean square error will score the forecast method, not only in this type of forecasting tool but also will be the tool to compare with other forecast methods. On the other hand, the dataset split is useful to compare the forecast and the observation in the test dataset and used the train dataset as the part that will feed the mathematical equations. The functions created are the following ones:

Figure 10: RMSE and split code

```
#CHECK THE ERROR PRODUCED IN THE TEST DATASET WITH THE RMSE
def measure_rmse(actual, prediction):
        return sqrt(mean_squared_error(actual, prediction))

#FIT THE MODEL DIVIDING THE DATASET IN TRAIN AND TEST SETS
def split_dataset(data, num_test):
        return data[:-num_test], data[-num_test:]
```

It is noteworthy that the root means square error, the dataset split and the following steps are harnessed in future methods.

Secondly, a function that iterates among the different possibilities called ('walk-forward-validation') is done. This function used the split function and create the history just by taking the training dataset. The next step is to forecast the next value and add it to the history. If these values are the ones used to predict future values, they will carry errors from last forecasted values that are important to take into account. However, in these models, the predictions use the real values. Finally, the function return the error which is the key element to score the model.

Figure 11: Walk forward validation

```
def walk_forward_validation(data, num_test, config):
  global predictions
  predictions = list()
  # split dataset
  train, test = split_dataset(data, num_test)
  # feed history with training dataset
  history = [x for x in train]
  for i in range(len(test)):
        # fit model and make forecast for history
        prov = simple_forecast(history, config)
        # store forecast in list of predictions
        predictions.append(prov)
        # add actual observation to history for the next loop
        history.append(test[i])
  # estimate the error
  error = measure_rmse(test, predictions)
  return error
```

The next function is responsible for scoring the model and showing all the results in the console. To summarise properly the results, it disregards not possible configuration methods. Moreover, as there are always warning messages in the console when running the code, this function will catch all the warnings messages and it will hide them in order to see better the results[18]. It is important to know this fact of the function because if there are errors in the code, after running it is not possible to check why it is not working, therefore sometimes is useful to disable this characteristic of the function.
Last task of the function is to display the results for each valid configuration.

Figure 12: Score model code

```
def score_model(data, num_test, config, debug=False):
  result = None
  # convert configuration to a string
  string = str(config)
  # show all warnings and fail on exception if debugging
  if debug:
        result = walk_forward_validation(data, num_test, config)
  else:
        # one failure during model validation means an unstable conf.
        try:
          # never show warnings when grid searching because it is noisy
          with catch_warnings():
                filterwarnings("ignore")
                result = walk_forward_validation(data, num_test, config)
        except:
          error = None
  # check for an interesting result
  if result is not None:
        print(' > Model[%s] %.3f' % (string, result))
  return (string, result)
```

The following function will be created to **sort the different results** paying attention to the root mean square error. The function compiles all the available results of the list of configurations and sorts it in order to know which configuration mode is the most useful one. But this function has an additional tool to speed up the program. This tool is the Parallel function, which is integrated into the joblib library. This tool checks the number of CPUs available in the computer and will use all of them so different calculations will be done avoiding bottlenecks[18].

Figure 13: Grid search code

```
def grid_search(data, config_list, num_test, parallel=True):
  scores=None
  if parallel:
        # executor is used to speed up the grid searching after
        # evaluating the number of cores of the computer
        executor=Parallel(n_jobs=cpu_count(), backend='multiprocessing')
        # define tasks (score_model() function for each configuration)
        tasks = (delayed(score_model)
          (data, num_test, config) for config in config_list)
        # execute list of tasks
        scores = executor(tasks)
  else:
    # execute tasks sequentially
        scores = [score_model
          (data, num_test, config) for config in config_list]
  #remove empty results
  scores = [r for r in scores if r[1]!=None]
  #sort configurations by error, asc
  scores.sort(string=lambda tup: tup[1])
  return scores
```

Last function of this forecast model is a simple loop that creates arrays of the **different configurations** possible in this problem, with different offsets, different n last values and different average types (mean, median or persist).

```
def simple_configurations(max_len, offsets=[1]):
  configurations=list()
  for i in range(1, max_len+1):
        for o in offsets:
          for t in ['persist', 'mean', 'median']:
                config=[i, o, t]
                  configurations.append(config)
  return configurations
```

All these functions are called by other functions doing a cascade process which the last functions explained are the first functions to be run.
Sometimes, it is not possible to run the code without using a preamble to avoid problems with it, depending on the operating system and the version of the tools, which is:

```
if __name__ == '__main__':
```

To run this code, some parameters will be defined to organise the dataset in order to obtain the results of one step forecast. First, the dataset will be read in time-series format using pandas library. Next code, shows the data input of the code.

```
                      energy (kWh)
tstp
2012-10-12 01:00:00          0.000
2012-10-12 02:00:00          0.000
2012-10-12 03:00:00          0.000
2012-10-12 04:00:00          0.000
2012-10-12 05:00:00          0.000
```

```
...                                   ...
2014-02-27  20:00:00                 2.372
2014-02-27  21:00:00                 1.480
2014-02-27  22:00:00                 0.899
2014-02-27  23:00:00                 2.597
2014-02-28  00:00:00                 2.605
```

These values, as it said above, are transformed in per unit values.

```
val_max=series.max()
series=series/val_max
```

Then, test dataset will be consider as the 100 last value of the real dataset. The offset component will be set as 1 to check the values obtained considering no seasonal component in the time series.
Then the functions for grid searching will be called to start searching for the best configuration for this case as can be seen in the following code:

```
series = read_csv('final_dataframe.csv', header=0, index_col=0)
del series['temperature (C)']
val_max=series.max()
series=series/val_max
data=series.values
# to split the dataset
num_test = 100
max_len = len(data) - num_test
config_list = simple_configurations(max_len)
scores = grid_search(data, config_list, num_test)
# top 3 configurations
for config, error in scores[:3]:
  print(config, error)
```

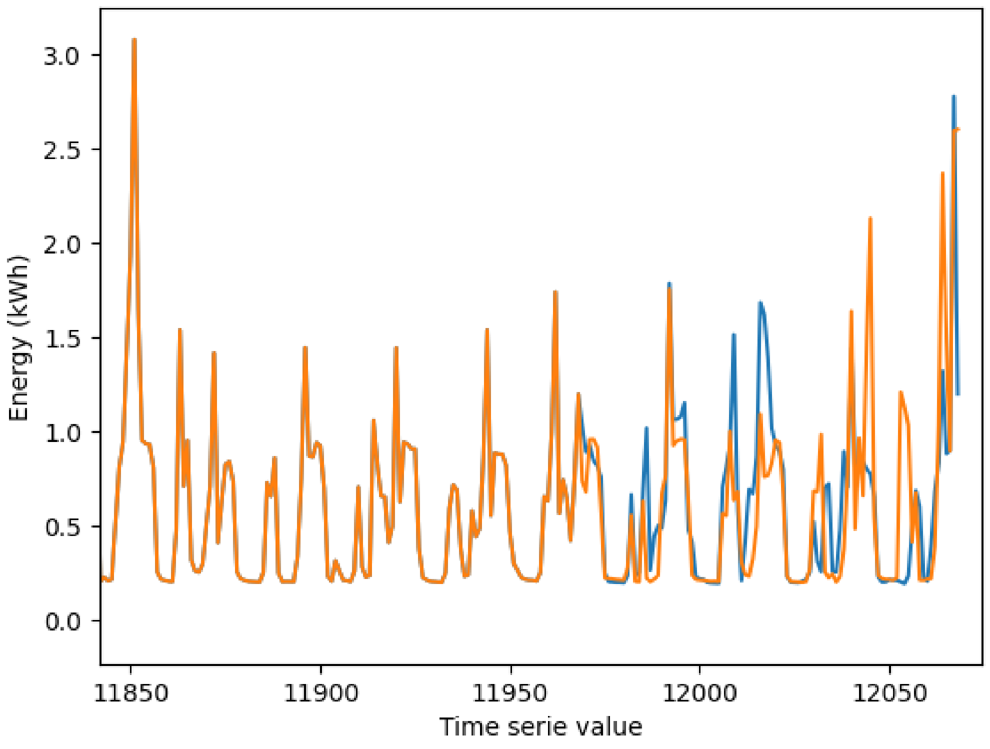
This code will give the best configurations for the forecast prediction model created, which is a basic model that **can be used to compare** the future models to know how good or bad are compared with this one.

In the first case, the London scenario with an energy shape that can be seen in figure 8, the best configurations, i.e. the closest one for obtaining the real values, are the next ones:

```
[1175, 1, 'persist'] 0,0794
[24, 1, 'persist'] 0,08676
[72, 1, 'persist'] 0,087334
```

The three best configurations are done by the **persist methods**. The best one uses **1175 n-values** without any seasonality parameter which seems to be the most accurate one. However, that could be just in the case of the 100 last values of the dataset, which is the test period used in this case. In figure 14 the results of the prediction are depicted, being the blue line the predictions and the orange line the real values.

Figure 14: Prediction of energy demand in London (persist method)



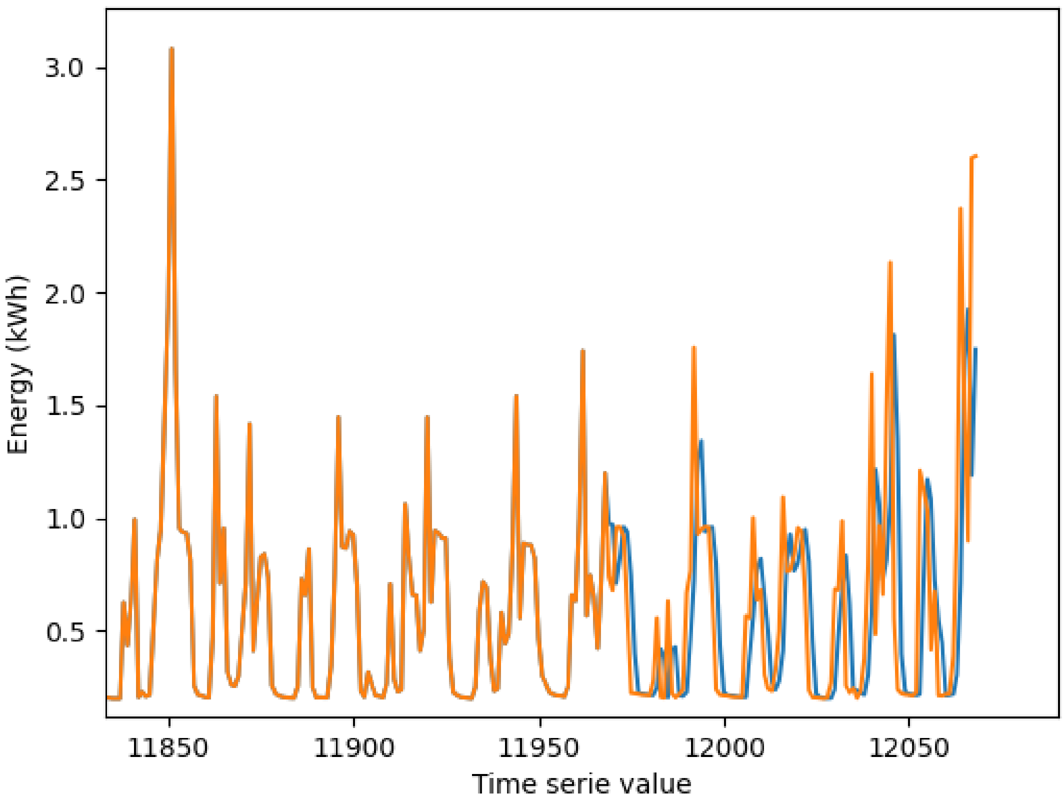
As can be seen in figure 14, the predicted values are not bad at all with a root mean square error of **0,0794** in p.u. value. The blue line shows the prediction and the orange line is the real observation. The difference between the two signals will be considered as how accurate the model behaves and will be measured by the RMSE.

Beyond the persist method, which can be not as extrapolated as the mean or median method, the best configurations modes are the following ones:

```
[2, 1, 'mean'] 0,09568
[2, 1, 'median'] 0,09568
[3, 1, 'mean'] 0,09787
```

As can be seen, the mean and median values of close previous observations are the best configurations that can be used to estimate future values not assuming seasonal components in the model, which makes sense. Using the best configuration method beyond the persist method, i.e. a **mean** predicted tool of the **2 previous values** with a RMSE of **0,096**, the forecast series will be as follow:

Figure 15: Prediction of energy demand in London (mean method)

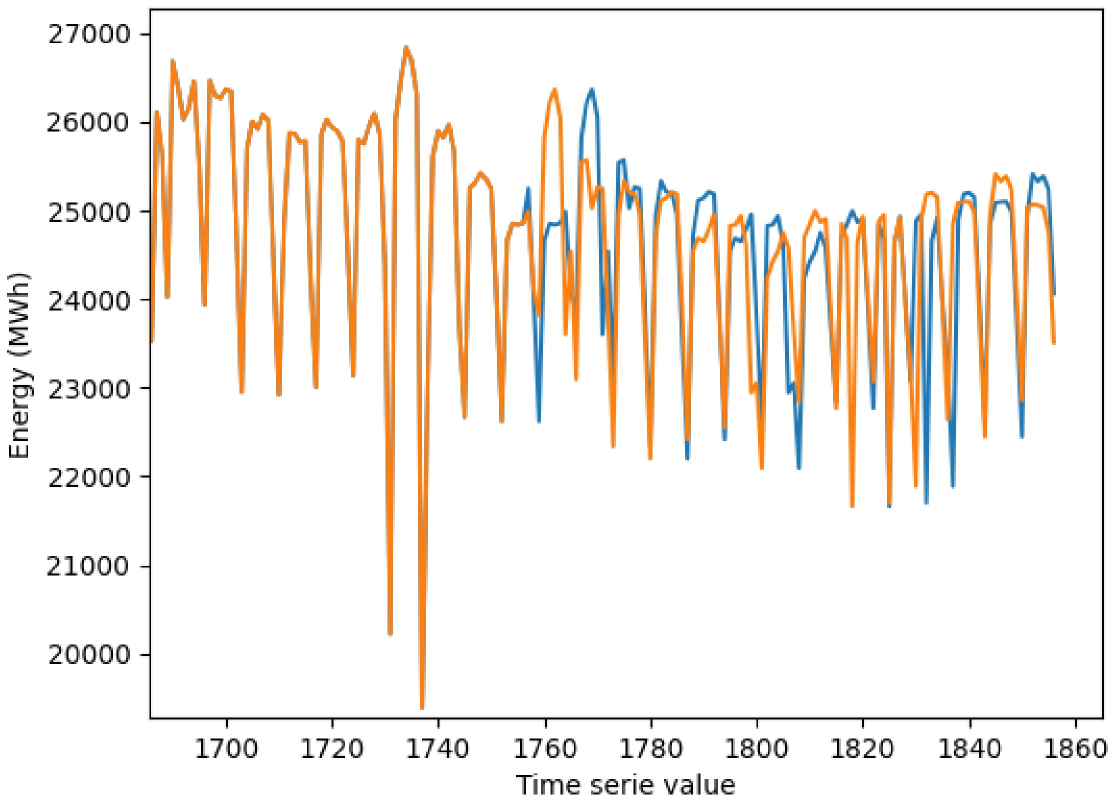


Nevertheless, in this time-series scenario is a little bit complicated to set a seasonal component through the stochastic behaviour of the signal. In a large amount of data as the Canarian energy demand, that can be appreciated in figure 9, it is possible to set an offset of 365 assuming a time step of one day and a **yearly seasonal behaviour**.
After running the code, it is possible to see that the second-best method uses a seasonal component that simplifies the calculations and makes it faster with a low RMSE in comparison to the rest of the errors.

```
[7, 1, 'persist'] 0,0308
[7, 365, 'persist'] 0,0308
[1106, 1, 'persist'] 0,0318
```

The best configuration in this case is a **persist** method of the **seven previous observation** but **without a seasonal component** with a RMSE of **0,0301**, that can be seen in figure 16.

Figure 16: Prediction of energy demand in Canarias (persist method)

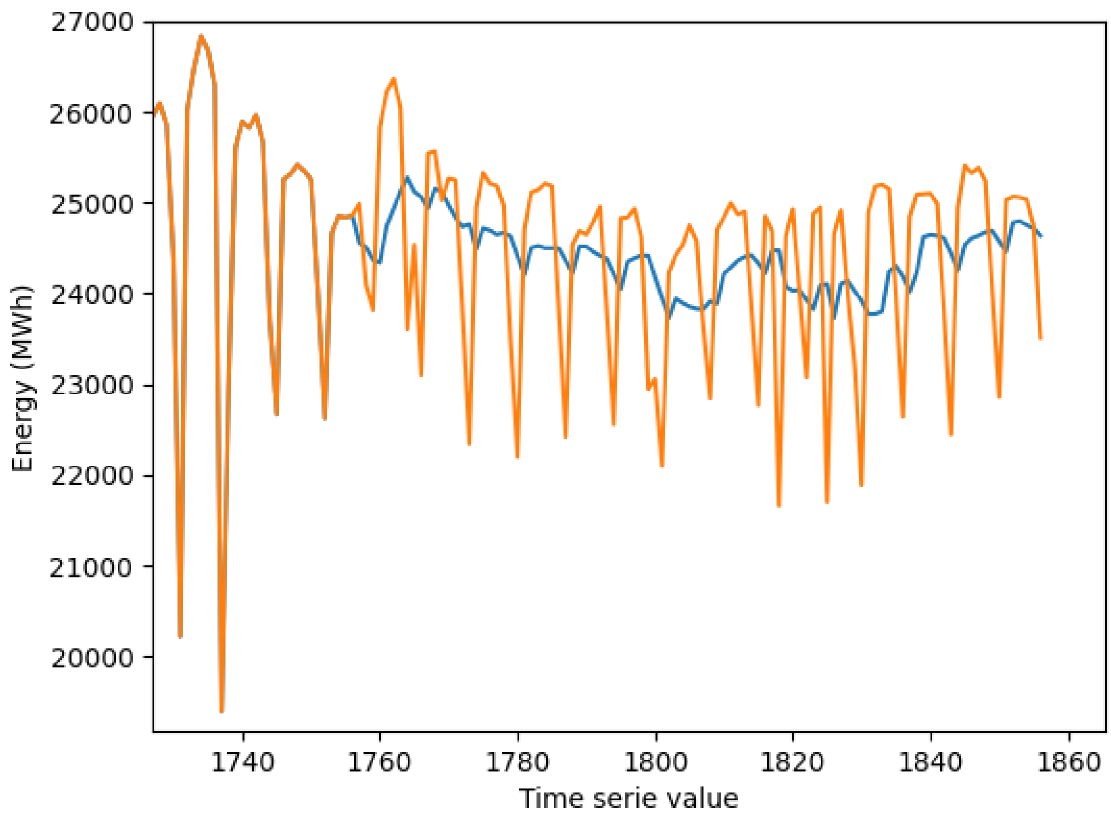


In this case, the output signal follows the trend of the real one without the seasonal component. After omitting the persist method, the top three configurations methods obtained are the following ones:

```
[8, 1, 'mean'] 0,03665
[22, 1, 'mean'] 0,03672
[29, 1, 'mean'] 0,03673
```

The result is a **mean** forecast tool of the previous **eight observations** with a RMSE not much bigger than the persist methods. As can be seen in figure 17, the model is still accurate.

Figure 17: Prediction of energy demand in Canarias (mean method)



It is possible to check that in mean predicted models, the output signal is smoother than the persist method.

5.3.2 Exponential smoothing model

This method uses some functions created also for the naive model as the grid searching function or the score function which aims to find the best configuration method for the model. In this case, the configuration function changes to fit our model. Moreover, a new forecast function is created that contains all the parameters related to the exponential smoothing that can be seen in section 4.1.1, page 15.
Using this forecast method, the trend is easier to detect. For instance, the next dataset:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

with a simple configuration for the exponential smoothing model, will show a future value of 11, meaning that the prediction is really accurate in linear trend scenarios. This model is much better for this type of dataset in comparison to the naive model because it can show values that are not included between the past values, something that is needed in these kinds of cases and can not be achieved with the naive model.

The forecast function for the exponential smoothing integrates all the parameters related to the model, which are the trend, the damping trend, the seasonal period and the seasonal trend. Moreover, a parameter known as use-boxcox informs about the utilization of the Box-cox function for the forecast. The input of the function is the time series and the configuration of the exponential smoothing. Beyond that, to fit the model, an optimization process for choosing the best options for values not set previously and a remove-bias tool that fit the values before being returned, are set.

```
def exp_smoothing_forecast(history, configuration):
  t,d,s,p,bo,b = configuration
  history = array(history)
  model = ExponentialSmoothing(history, trend=t, damped_trend=d,
    seasonal=s, seasonal_periods=p, use_boxcox=bo)
  model_fit = model.fit(optimized=True, remove_bias=r)
  prov = model_fit.predict(len(history), len(history))
  return prov[0]
```

The function, as happens with the naive model, gives back the prediction that follows the trend of the model because of previous observations.

The root mean square error and the dataset split done in the naive model (figure 10) is also integrated in this model, as well as the walk forward validation (figure 11), the score model function (figure 12) and the grid search function (figure 13).

Beyond this functions, a function that creates all possible configuration methods of the exponential smoothing model to select the best one is created.

```
def exp_smoothing_configurations(seasonal=[None]):
  models = list()
  # define the configuration lists
  trend_p = ['add', 'mul', None]
  damping_p = [True, False]
  seasonal_p = ['add', 'mul', None]
  # seasonal=[None] by default
  #perior_p must integrate the seasonal parameter of the model
  perior_p = seasonal
  boxcox_p = [True, False]
  bias_p = [True, False]
  # create configuration instances
  for t in trend_p:
        for d in damping_p:
          for s in seasonal_p:
                for p in perior_p:
                  for bo in boxcox_p:
                        for b in bias_p:
                          config = [t,d,s,p,bo,b]
                          models.append(config)
  return models
```

As happen before, an input conditions will be coded in order to call the function and organise the data to study. This part set the test series as the last 100 values as it happen in the naive model and call the function for searching the best configurations.

```
if __name__ == '__main__':
  series = read_csv('final_dataframe.csv', header=0, index_col=0)
```

```
del series['temperature (C)']
val_max=series.max()
series=series/val_max
data = series.values
# split dataset
num_test = 100
max_len = len(data) - num_test
# model configurations without seasonal as default
config_list = exp_smoothing_configurations()
scores = grid_search(data[:,0], config_list, num_test)
# top 3 configurations
for config, error in scores[:3]:
      print(config, error)
```

After running the code, the result of the best configuration methods are the following ones for the **London** scenario:

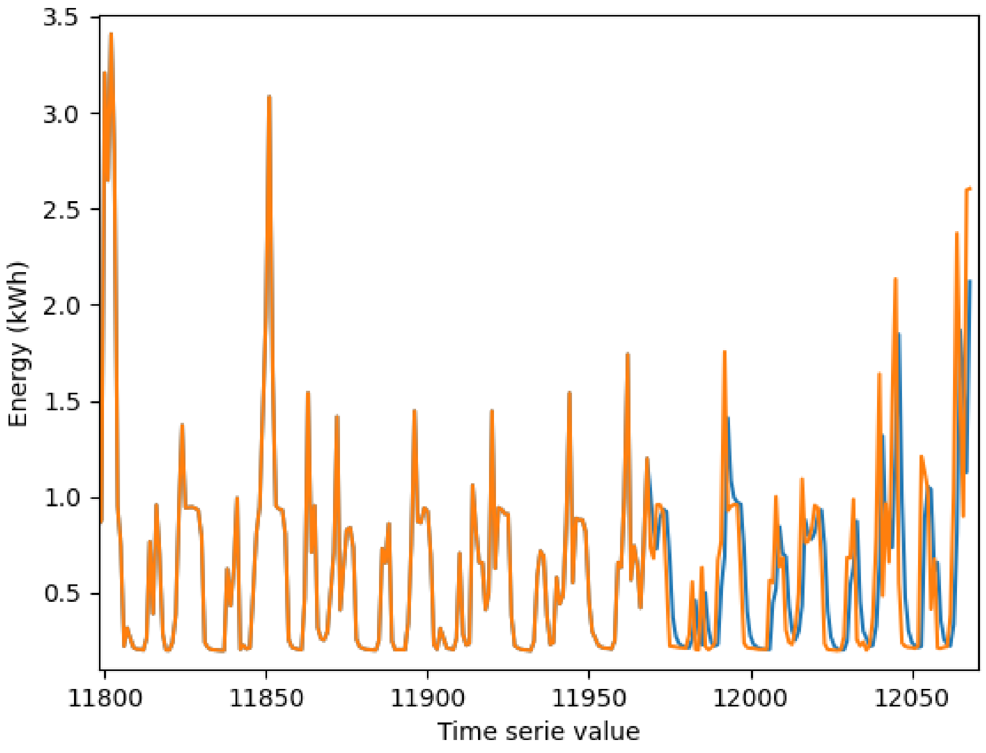
```
[None, False, None, None, False, False] 0,09015109
['add', False, None, None, False, False] 0,0901512
['add', True, None, None, False, False] 0,0901514
```

The results mean:

```
[Trend, Dampening, Seasonal, Seasonal period, Box_cox, Bias] RMSE
```

The code created informs that the best configuration is created by **neglecting the trend** and **without a damping parameter** that changes the influence of the trend over time. Moreover, it **does not include a seasonal component** and therefore avoids any information about the period of the season. That was because the code was set by default to neglect seasonality conditions. Finally, avoid the use of the Box-cox function and neither remove the bias.
The results of the model can be seen in figure 18.

Figure 18: Exponential smoothing in London scenario

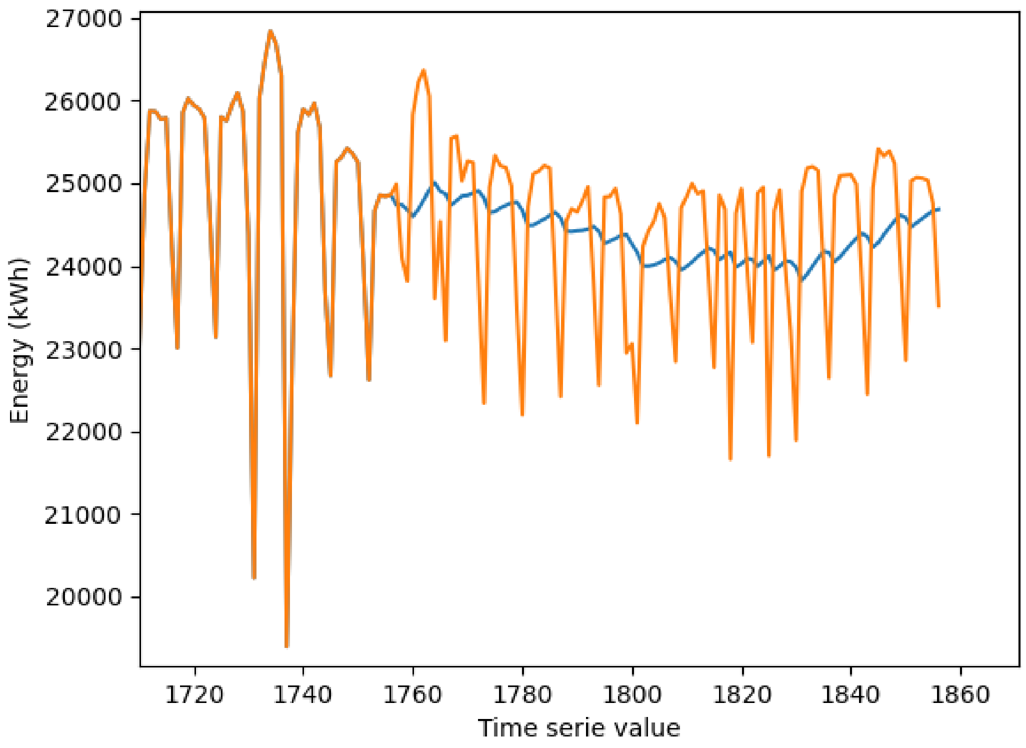


On the other hand, after searching for the best configuration method in the case of Canarias' scenario, including the seasonal component of 365 values, i.e. a year seasonal period, the best configuration methods are the following ones:

```
['add', True, None, 0, True, False] 0,037578
['add', True, None, 365, True, False] 0,037578
['add', True, None, 0, True, True] 0,037584
```

As can be seen, the second-best option even with a seasonal period set, the seasonal trend is deactivated, meaning that **no configuration will recommend adding the seasonal component**. As in the London scenario, the **additive trend** is the best configuration with a **damp component** that changes over time. Moreover, the best configuration, beyond disregarding seasonal components, which is not easy to predict due to the behaviour of the data, uses the Box-cox configuration but doesn't use the bias-remove tool. Plotting this configuration shows the following prediction.

Figure 19: Exponential smoothing in Canarias scenario



5.3.3 SARIMA model

The SARIMA model, like the previous ones, reuses some parts of the code used for searching for the best configurations as the code for calculating the RMSE and the dataset split (figure 10), the walk-forward validation (figure 11), the score model function (figure 12) and the grid search function (figure 13).

SARIMA model, as it is mentioned in section 4.1.2 in page 18 uses an autoregressive (AR) parameter, a moving average (MA) parameter and an integrated (I) parameter.
As it happens with the exponential smoothing, the fact of paying attention to the trend makes the possibility of predicting values that are still out of the dataset range.
To know which are the more favourable parameters, tools such as the autoregressive function, the partial autoregressive function or the Augmented Dickey-Fuller test can help to determine the best values.
Although the idea is to use a grid search function that will determine the parameters, all these tools are still helpful.

The function that will predict the future values will be the next one:

```python
def sarima_forecast(history, configuration):
  order, sorder, trend = configuration
  model = SARIMAX(history, order=order, seasonal_order=sorder,
    trend=trend, enforce_stationarity=False, enforce_invertibility=False)
  model_fit = model.fit(disp=False)
  prov = model_fit.predict(len(history), len(history))
  return prov[0]
```

The code adds the order, the seasonal order and the dataset trend for each dataset to predict the future values after fitting the model, i.e. fit the model by maximum likelihood via Kalman filter, and setting the parameters.
To set this parameter, a list of all possible parameters will be introduce thanks to next code.

```
def sarima_configurations(seasonal=[0]):
  models = list()
  # define configuration lists
  p_params = [0, 1, 2]
  d_params = [0, 1]
  q_params = [0, 1, 2]
  t_params = ['n','c','t','ct']
  P_params = [0, 1, 2]
  D_params = [0, 1]
  Q_params = [0, 1, 2]
  m_params = seasonal
  # create configuration instances
  for p in p_params:
    for d in d_params:
      for q in q_params:
        for t in t_params:
          for P in P_params:
            for D in D_params:
              for Q in Q_params:
                for m in m_params:
                  config = [(p,d,q), (P,D,Q,m), t]
                  models.append(config)
  return models
```

The p, d, q, P, D and Q parameter uses only close previous values now but they can change depending on the model. The ['n', 'c', 't', 'ct'] trend option is for no trend, constant, linear, and constant with linear trend respectively.
All this code will make an easy way to know which parameters are the best ones for each scenario, that can be overcome with the help of other tools. For instance, after introducing a dataset as the following one:

```
[10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]
```

If the dataset is separated using the four last values to forecast and the previous to train the model, the best configuration would be this one:

```
[(1, 1, 0), (0, 0, 0, 0), 'c'] 0,0
```

Being the parameters the following coefficients:

```
[(p, d, q), (P, D, Q, m), trend] RMSE
```

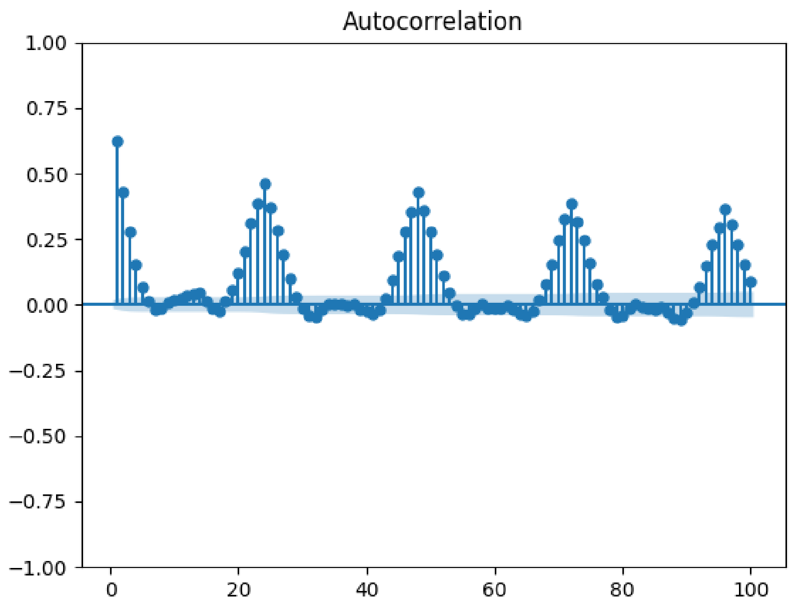
It is easy to realize that the system follows a linear trend and it doesn't include any seasonality, therefore the second list $(0, 0, 0, 0)$, doesn't include any parameter, showing a non-seasonality dataset. On the other hand, the p, d and q are $(1, 1, 0)$, showing that the last value is the one that matters for the AR parameter and neglecting the MA condition. With these characteristics, the error of the forecast model is 0, showing a perfect model for this condition.

This code **takes much more time** than the previous ones. In fact, one possibility to reduce the computational process is to trim the dataset in order to speed up the process.
However, for this method and some others, the **supercomputer** of the UPC department was used, in order to reduce the execution time. It is important to take into account the limitations of the computer which runs previous codes.

For the London scenario, after doing the Augmented Dickey-Fuller test[1] , the p-value given is $0,084$ which is bigger than $0,05$, meaning that the stationary condition is not fulfilled. To transform to a stationary time-series, one integration process will be expected.

On the other hand, the autocorrelation function informs about the expected moving average value, which in the case of simple scenarios is easy to detect. however, if the model is made up of a long dataset is difficult to guess as can be seen in figure 20 which shows the ACF of the London scenario.
Nevertheless, is possible to see a trend in the spikes of the figure with ≈ 25 time steps.

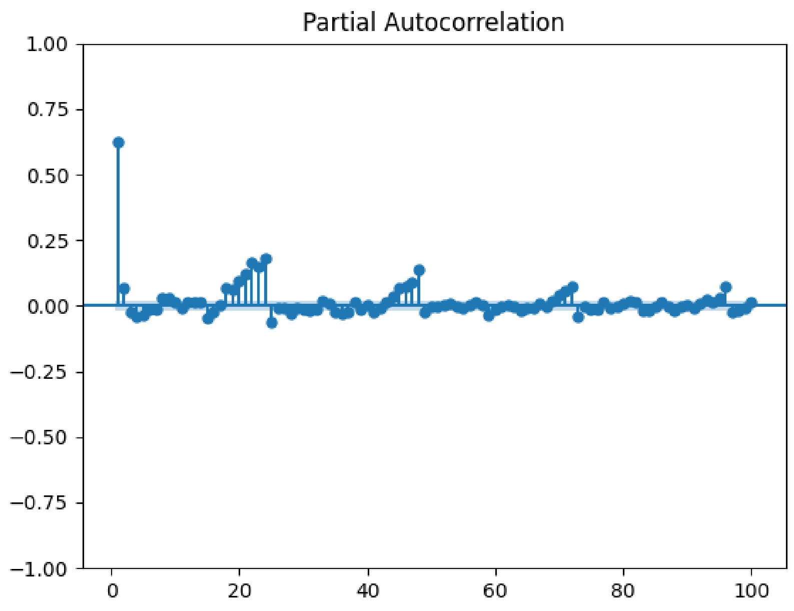
Figure 20: Autocorrelation function for the London scenario



Moreover, the PACF shows values related with $AR(p)$, depicting also a trend in the spikes of the charts with the same range of the ACF approximately.

[1]To know more about the Augmented Dickey-Fuller test, check section 4.1.2 of page 18

Figure 21: Partial Autocorrelation Function for the London scenario

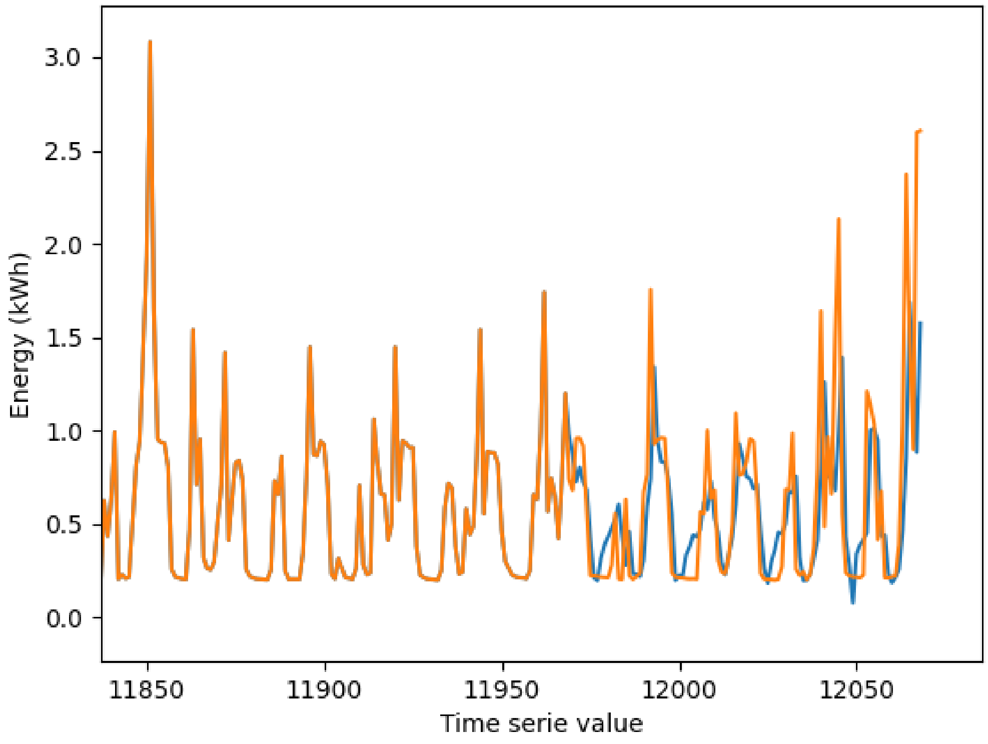


In this scenario, the seasonality condition is not clear, meaning that the P, D, Q and m coefficient will be set at 0. After running the code, the top three configurations are the following ones:

```
[(6, 0, 6), (0, 0, 0, 0), 'c'] 0,08331
[(5, 0, 4), (0, 0, 0, 0), 'c'] 0,08353
[(6, 0, 5), (0, 0, 0, 0), 'c'] 0,08364
```

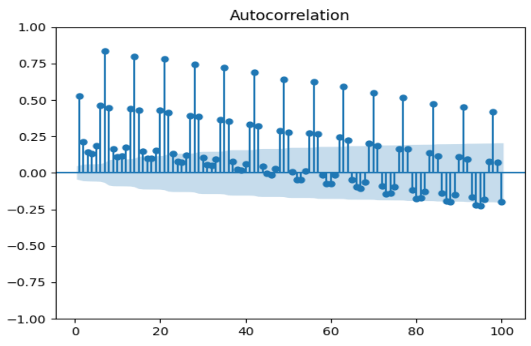
The first case uses $MA(q) = 6$ which is the highest spike of the PACF, and the $AR(p)$ parameter is 6, which is also the highest spike of the ACF result.
The stationary condition is not assumed and their related parameters (P, D and Q) are neglected.
On the other hand, the selected trend for the model is a constant trend.
The result of the best configuration forecast for the last 100 values can be seen in figure 22.

Figure 22: SARIMA forecast in London scenario

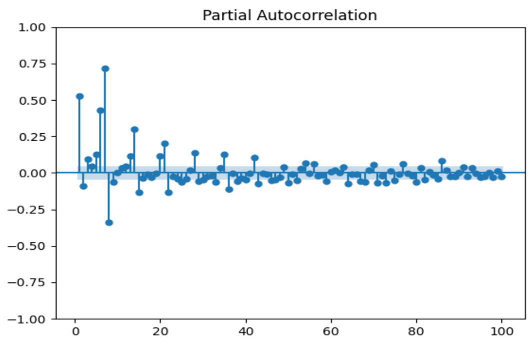


In this case, the **RMSE** for the 100 last values forecast is **0,08331**, which still is not better than the result of the naive model.

On the other side, in the **Canarias** scenario, after checking the ACF of figure 23a it is possible to size the configuration loop. In the figure of the ACF, which is useful to determine q (moving average parameter), there is a seasonality where the first spike is founded in the seventh position. Although there are more significant spikes in the chart, to reduce the complexity of the model, they will be omitted and the first spike will be considered the significant one. Moreover, the PACF of figure 23b, which is useful to size the p parameter, shows also a seasonality on it, having also a big spike in the seventh lag.



(a) Autocorrelation function



(b) Partial autocorrelation function

Figure 23: ACF and PAFC

Then, the d parameter related to the integrated parameter depends on the Augmented Dickey-Fuller test which determines its nature. After testing the dataset, the p-value obtained was 0.600 which is bigger than 0.05 but just **after one integration**, the value decreases to $5.274 \cdot 10^{-14}$. A **d equivalent to 1** is enough to transform to a stationary dataset[2].

Finally, the configuration loop ends as follows:

```
def sarima_configurations(seasonal=[0]):
  models = list()
  # define configuration lists
  p_params = [0, 1, 2, 3 ,4 ,5 ,6 ,7]
  d_params = [0, 1]
  q_params = [0, 1, 2, 3, 4, 5, 6, 7]
  t_params = ['n','c','t','ct']
  P_params = [0, 1, 2]
  D_params = [0, 1]
  Q_params = [0, 1, 2]
  m_params = seasonal
  # create configuration instances
  for p in p_params:
    for d in d_params:
      for q in q_params:
        for t in t_params:
          for P in P_params:
            for D in D_params:
              for Q in Q_params:
                for m in m_params:
                  config = [(p,d,q), (P,D,Q,m), t]
                  models.append(config)
  return models
```

A seasonal component of 7 days is also added, which is a variation with last models:

```
config_list  = sarima_configs(seasonal=[0, 7])
```

The top three configurations after the conditions were determined are the next ones:

```
[(7, 1, 7), (0, 0, 0, 0), 'c'] 0,02706
[(7, 0, 7), (0, 0, 0, 0), 'c'] 0,02711
[(7, 0, 7), (0, 0, 0, 0), 'n'] 0,02719
```

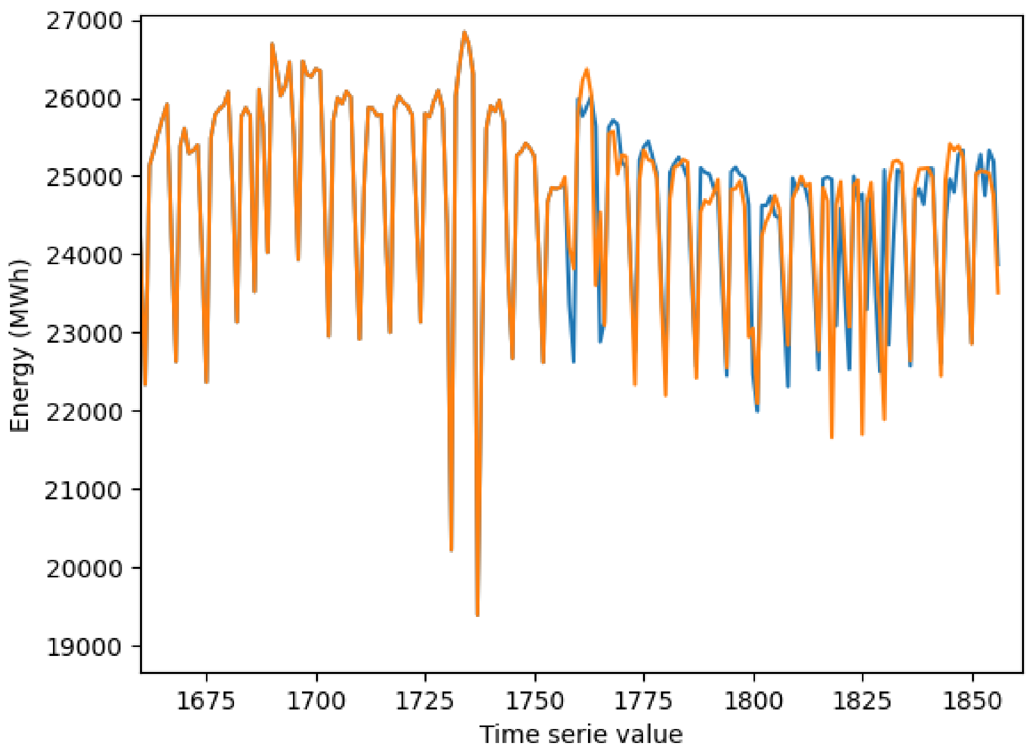
The best configuration in this case is a $AR(p) = 7$ and $MA(q) = 7$. The AR and the MA parameter could be seen in the PAFC and the AFC.
One curious thing in these results is that **none of them consider the seasonality** of the dataset in order to do the predictions more accurate.
Moreover, all the results omit the P, D and Q terms, because they are working without seasonality on it.
Finally, the 'c' trend corresponds to a **constant trend**.

The forecast done by the best configuration of the last 100 values with previous values to train

[2]To better understand this test, in section 4.1.2 of page 18 is better explained.

the model can be seen in figure 24.

Figure 24: SARIMA in Canarias scenario



The **error** in this case is **0,0312**.

5.3.4 SARIMAX model

For the SARIMAX model, the configuration parameter used were the same as the SARIMA model.
For the London scenario

```
[(6, 0, 6), (0, 0, 0, 0), 'c']
```

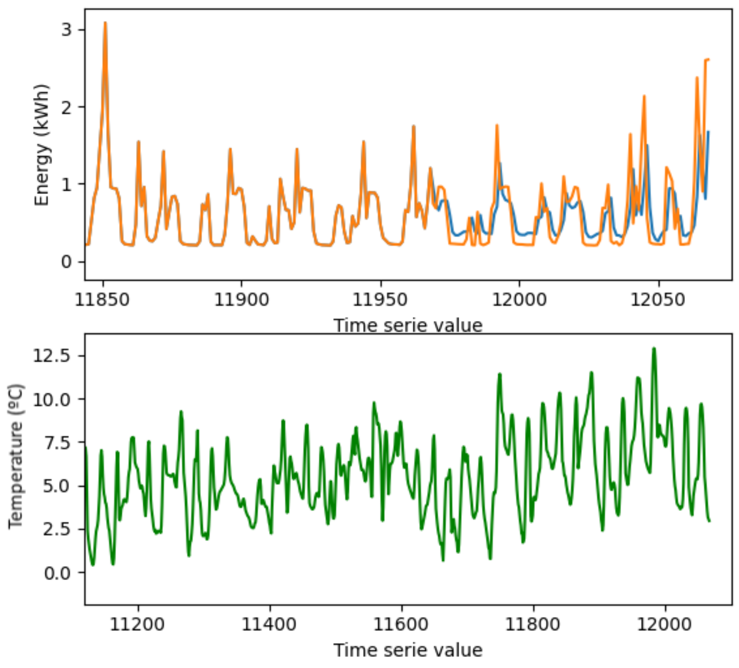
In the Canarias scenario, the configuration used is:

```
[(7, 1, 7), (0, 0, 0, 0), 'c']
```

The only difference between this method and the last one is the **addition of an exogenous variable** which is the **temperature** that tries to forecast the model more accurately. The addition of an exogenous variable means a variable that affects the model but it is not directly affected by it.

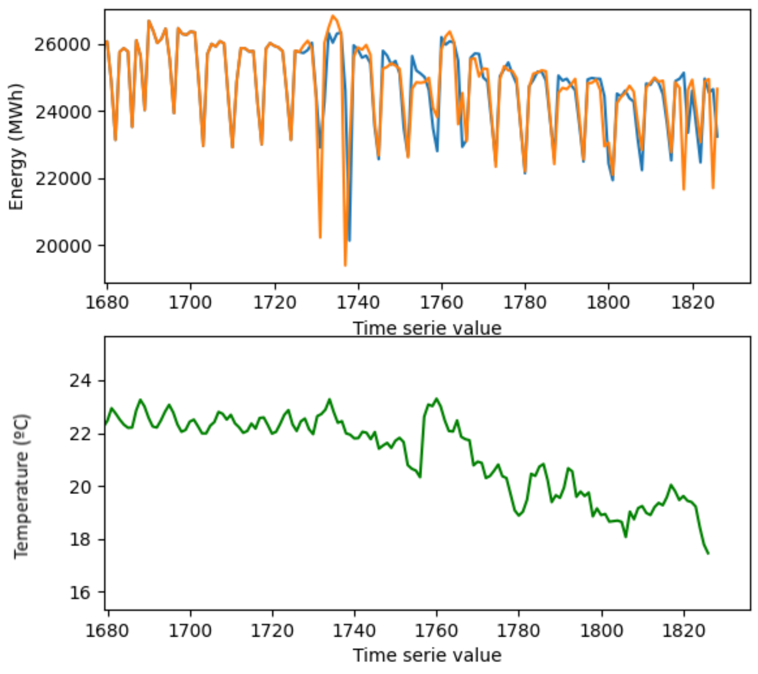
In the case of London, the forecast results obtained were not better than before. The accumulate RMSE for this scenario is **0,0874**, which is a bigger error than in the last case. The forecast graph can be seen in figure 25.

Figure 25: SARIMAX forecast for London scenario



On the other side, in the Canarias' scenario, the results were not either better than the SARIMA model. The accumulated RMSE is also bigger in this case, specifically **0,0381**. For this scenario, the forecast graph can be seen in figure 26.

Figure 26: SARIMAX forecast for Canarias scenario



5.3.5 Convolutional Neural Network (CNN) model

This will be the first neural network model done in this report. As was said above CNN was developed in order to recognise two-dimensional images but it also can forecast time series. The model uses some functions that will be repeated in future neural networks. These functions are the train-split function which separates the dataset into the train set to feed the model and the test set to score the model, the series_to_supervised in order to change the dimensions of the input data so it suits the one needed for some neural networks such as the CNN or LSTM, the walk-forward validation that scores the model as it happened with statistical methods and the repeat_evaluate function that tries to reduce the stochastic behaviour of the NN in order to score it better.

The train-split function is a simple function that separates the time series into two sets with the following code:

```python
def split_dataset(data, num_test):
  return data[:-num_test], data[-num_test:]
```

Then, a series_to_supervised will transform the data from a time series to a three-dimensional matrix. Moreover, the function must use a parameter that allows the user to set the number of lags studied in the model. As it was said above, the time series must be completed because if not some mistakes will be addressed. Therefore, if there are no missing values and it is possible to assure the frequency of the set, it is possible to delete the data-time column, which will be also done in this function.

```python
def series_to_supervised(data, n_in, n_out=1):
  df = DataFrame(data)
  cols = list()
  # input sequence (t-n, ... t-1)
  for i in range(n_in, 0, -1):
        cols.append(df.shift(i))
  # if it is more than one output...
  for i in range(0, n_out):
        cols.append(df.shift(-i))
  # put it all together
  agg = concat(cols, axis=1)
  # drop rows with NaN values
  agg.dropna(inplace=True)
  return agg.values
```

After, as it was done with statistical methods, the model must be scored and as it was done in the previous method, the tool that scores these models is the **RMSE**.

```python
def measure_rmse(actual, prediction):
  return sqrt(mean_squared_error(actual, prediction))
```

Then, the model must be prepared to predict the future values and this one will depend on the nature of each NN. As in this case, the NN is a CNN, the next function that it is called in order **to develop the model** will be as follow:

```python
def model_fit(train, configuration):
  n_input, n_filters, n_kernel, n_epochs, n_batch = configuration
  # prepare data
  data = series_to_supervised(train, n_input)
  train_x, train_y = data[:, :-1], data[:, -1]
  #reshape the model
  train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], 1))
  # The different layers of the CNN models are:
  model = Sequential()
  model.add(Conv1D(n_filters, n_kernel, activation='relu',
    input_shape=(n_input, 1)))
  model.add(Conv1D(n_filters, n_kernel, activation='relu'))
  model.add(MaxPooling1D())
  model.add(Flatten())
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch,
    verbose=0)
  return model
```

The model uses **two convolutional layers** with a **rectified linear unit** and a filter previously set. The number of filters means the number of parallel weights that the input layer will project[1]. On the other side, the **kernel size** defines the **number of time steps reads in each snapshot**. Then a **pooling layer** will reduce the complexity of the NN, reducing the input size by 1/4 and a **flatten layer** will link the pooling layer to the dense layer with a one-dimensional vector, used to make one-step predictions.
Then after fitting the NN, a prediction of the dataset will be carried out with the function model-predict. In the case of CNN, the model-predict function will look as follow:

```
def model_predict(model, history, configuration):
  n_input, _, _, _, _ = configuration
  # converto from 2D to 3D data
  x_input = array(history[-n_input:]).reshape((1, n_input, 1))
  # forecast
  prov = model.predict(x_input, verbose=0)
  return prov[0]
```

This prediction is added to a list which will be compared constantly to the actual values to know the accuracy of the model. This comparison will be done for each step of the prediction.

```
def walk_forward_validation(data, num_test, config):
  global predictions
  predictions = list()
  train, test = split_dataset(data, num_test)
  model = model_fit(train, config)
  history = [x for x in train]
  for i in range(len(test)):
        prov = model_predict(model, history, config)
        predictions.append(prov)
        history.append(test[i])
  # estimate the error
  error = measure_rmse(test, predictions)
  print(' > %.3f' % error)
  return error
```

The last step is a common function for all the NN which is the **reduction of the stochastic behaviour**, repeating the test, to use the mean RMSE. This step can be easily done by following code:

```
def repeat_evaluate(data, configuration, num_test, n_repeats=30):
  best = [walk_forward_validation(data, num_test, configuration)
    for _ in range(n_repeats)]
  return best
```

After running the code, maybe will be interested to summarize the most promising values. Unlike statistical methods, this time, the model will also calculate the mean and the standard deviation to have an idea of the spread of performance. This method will be carried out with the summarize_score method.

```
def summarize_best(name, best):
  best_m, score_std = mean(best), std(best)
  print('%s: %.3f RMSE (+/- %.3f)' % (name, best_m, score_std))
```

In this scenario, the variables set by the user will be the **number of inputs** or the **number of lags observations** used for each sample, the **number of parallel filters**, the **number of time steps** considered, the **training time** and the **number of samples**. The best thing to do is, actually a grid search to know which are the best hyper parameters, however, this analysis can take weeks or months to calculate with good computer processors. Therefore, as a first step, these values will be set after a trial and error process. It is worth saying that the variation of the results was not bigger than $\pm 0,01$. The input configuration can be seen in the last part of the code.

```
config=[50, 50, 3, 50, 50]
```

Which means:

```
config=[n_input, n_filters, n_kernel, n_epochs, n_batch]
```

This means that the **number of lags observations is 50**, there are **50 parallel filters**, **three time steps considered in each prediction**, **50 times that the model will be exposed** to the whole training and **50 samples within an epoch**.

The result of the prediction method in the **London scenario** it is shown in figure 27. The RMSE of this scenario was **0,081**.

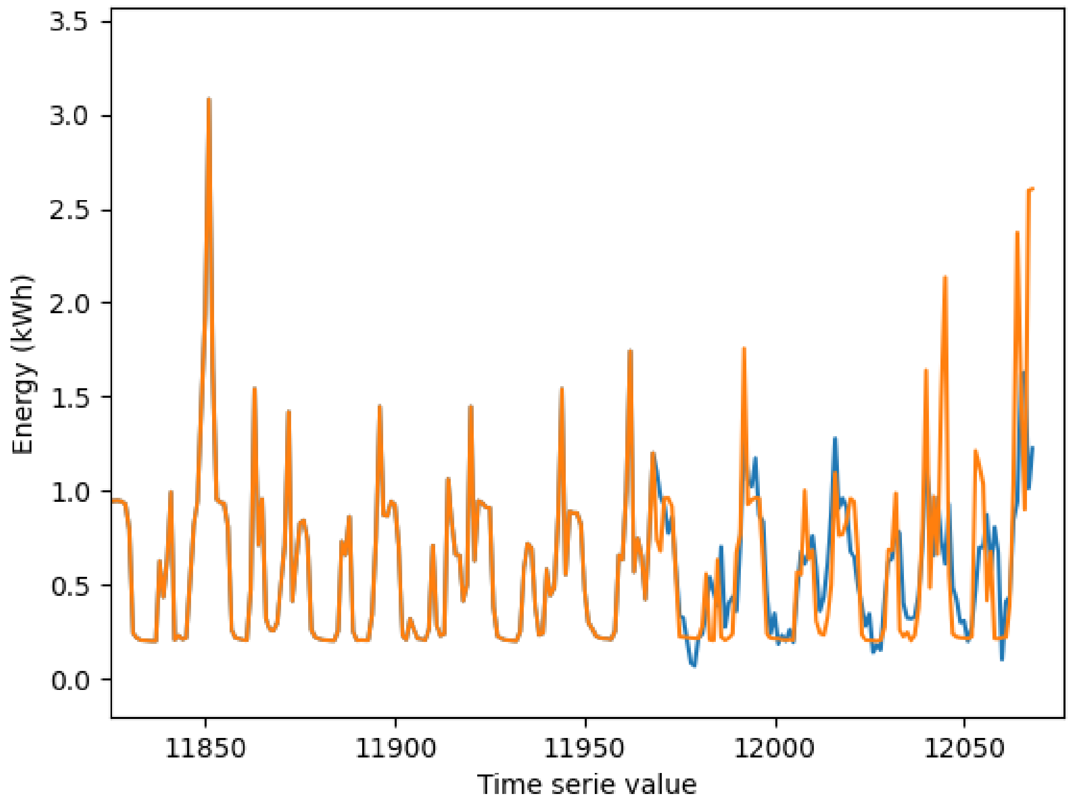


Figure 27: CNN prediction in London scenario

In the second scenario, the configuration used was determined also after a trial and error process, having the following configuration:

```
config=[100, 100, 5, 100, 100]
```

This means **100 lags observations** per prediction, **100 parallel filters**, **five time steps** considered, that the model was **exposed 100 times** to the whole training and **100 samples within an epoch**. After the trial and error, it was checked that the variation among different configurations was not really significant, having a final error of **0,028**.
The final prediction of this scenario can be seen in figure 28.

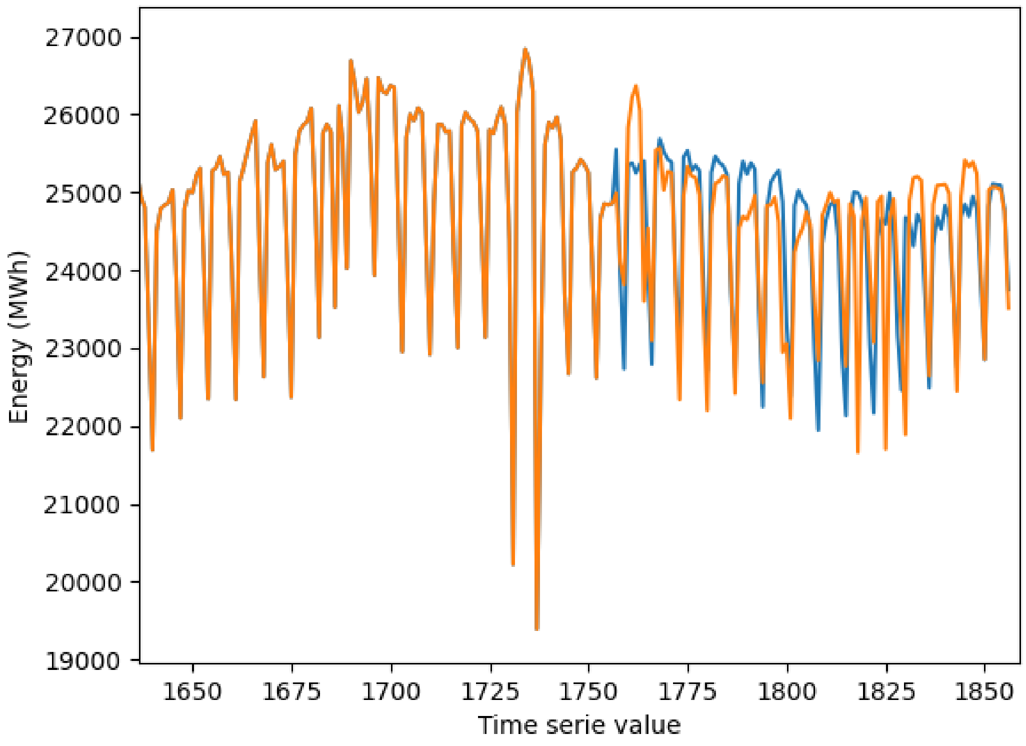


Figure 28: CNN prediction in Canarias' scenario

5.3.6 Recurrent Neural Network (RNN) univariate model

As was said above, there are different configurations of the LSTMs that can be carried on. In this report, the LSTMs that will be used is the **CNN-LSTM**, which uses **CNN to** manipulate subsequences and to **feed the LSTM** network **that will make the prediction**. Therefore, the data preparation of this model will be really similar to the last model due to the CNN is the part that manipulates the sequences again.

The aim of this NN is to automatically learn and gives the proper data from the existing dataset without manipulating it in order to suit it better.
The samples (input and output samples) will be divided into different subsequences and different time steps, that the CNN will use as an input and after the computational process, it will provide the proper information to the LSTM neural network.
If there is a monthly dataset of 2 years it is possible to divide the model into four subsequences of six time steps each.

As it was be mentioned before, the train-split function, the series_to_supervised, the walk_forward_validation and the repeat_evaluate functions will be the same as the last NN or really similar.
The main different to the last NN is the change of the model_fit() function that follow the next code.

```
def model_fit(train, configuration):
  n_seq, n_steps, n_filters, n_kernel, n_nodes, n_epochs, n_batch = configuration
  n_input = n_seq * n_steps
  data = series_to_supervised(train, n_input)
  train_x, train_y = data[:, :-1], data[:, -1]
  train_x = train_x.reshape((train_x.shape[0], n_seq, n_steps, 1))
  # layers of the model:
  model = Sequential()
  model.add(TimeDistributed(Conv1D(n_filters, n_kernel, activation='relu',
  input_shape=(None,n_steps,1))))
  model.add(TimeDistributed(Conv1D(n_filters, n_kernel, activation='relu')))
  model.add(TimeDistributed(MaxPooling1D()))
  model.add(TimeDistributed(Flatten()))
  model.add(LSTM(n_nodes, activation='relu'))
  model.add(Dense(n_nodes, activation='relu'))
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model
```

In this model, the CNN model is configured through the next lines:

```
# CNN model
model = Sequential()
model.add(TimeDistributed(Conv1D(n_filters, n_kernel, activation='relu',
  input_shape=(None,n_steps,1))))
model.add(TimeDistributed(Conv1D(n_filters, n_kernel, activation='relu')))
model.add(TimeDistributed(MaxPooling1D()))
model.add(TimeDistributed(Flatten()))
```

The same CNN must be applied to each input of the subsequence, which will be an input variable of the model. Therefore **TimeDistributed layer** appears, to **use the same** one for the **different subsequences**. It can be seen that there is also a **convolutional layer** with its specific number of filters and kernel and a **rectified linear unit as the activation function**. This is followed by a **pooling layer** that simplifies the system and a **flatten layer** that suits the information to the LSTM later. The output of each CNN submodel will be a vector. This time series will be the input of the LSTM model which is responsible to make the predictions.

The input parameters of the model that must be set by the user are the **number of subsequences** per sample, **number of time steps** of each subsequence, the **number of parallel filters** of the CNN, the **number of time steps** considered in each reading of the input subsequence for the CNN, the **number of LSTM nodes** of each hidden layer, the **number of times** that the **model must train** and the **number of samples within an epoch**.

The model_predict() function also has some changes this time in order to split each sequence into subsequences. The function will be written as follow:

```
def model_predict(model, history, configuration):
  n_seq, n_steps, _, _, _, _, _ = configuration
  n_input = n_seq * n_steps
  x_input = array(history[-n_input:]).reshape((1, n_seq, n_steps, 1))
  prov = model.predict(x_input, verbose=0)
  return prov[0]
```

The part of the function that suits the vector array is the next one:

```
x_input = array(history[-n_input:]).reshape((1, n_seq, n_steps, 1))
```

As happened before, due to the high computational cost of grid searching in this kind of prediction method, the methodology used is a try-error process that doesn't guarantee the best configuration but can give an idea of the final result. The input parameters are in the last part of the code.

```
config=[4, 24, 64, 5, 100, 200, 100]
```

where:

```
config=[n_seq, n_step, n_filters, n_kernel, n_nodes, n_epochs, n_batch]
```

Although the configuration was not found by a grid search, during the try-error process, the results obtained were not very different from each other. The configuration selected means that there are **4 subsequence numbers within a sample**, that there are **24 time steps in each subsequence**, **64 parallel filters**, **5 time steps** considered **in each read process**, **100 LSTM units** in the hidden layer, **200 times that the model will be exposed** to the training dataset and **100 samples within an epoch**.
The result of this configuration gives, as an average, an $RMSE = 0,078$. The result of the model can be seen in figure 29.

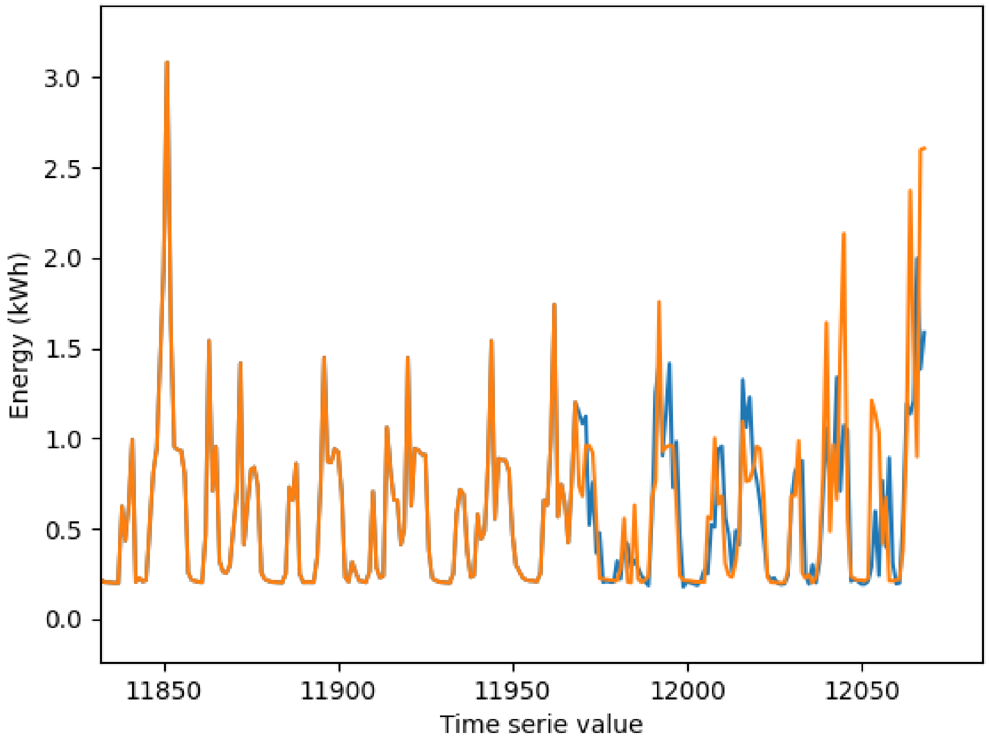


Figure 29: CNN-LSTM forecast in London scenario

On the other hand, in the second scenario, the configuration was also found by a try-error process and, as happened before, the different results were really close.
The final configuration used in this scenario was:

```
config=[3, 12, 64, 3, 100, 200, 100]
```

This configuration means that there are three subsequence numbers within a sample, that there are **12 time steps** in **each subsequence**, **64 parallel filters**, **three time steps considered in each read process**, **100 LSTM units** in the hidden layer, **200 times that the model will be exposed** to the training dataset and **100 samples within an epoch**.
The result of the model gives a root mean square error of **0,028** this time and the forecast made can be seen in figure 30.

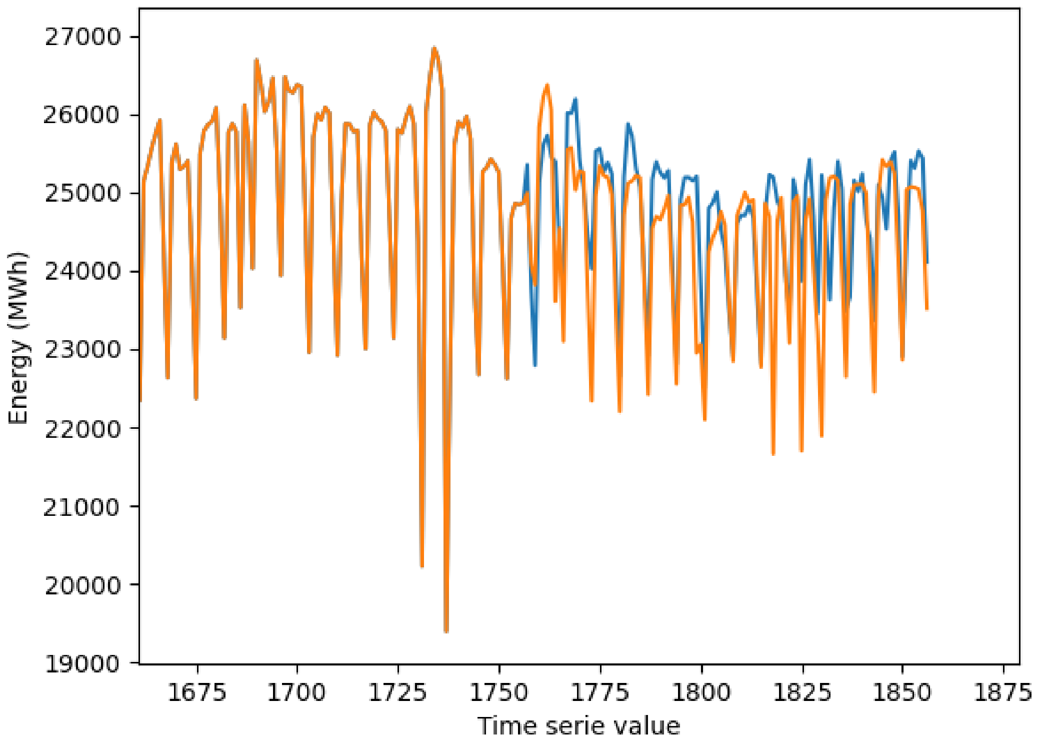


Figure 30: CNN-LSTM forecast in Canarias scenario

5.3.7 Recurrent Neural Network (RNN) multivariate model

This section of **multivariate** and **multistep** analysis with a NN, i.e. feed the system with multiple inputs and gives the forecast of future values at one time, is done in order to score how ANN behaves with more than one variable. Moreover, the **economic impact** can also be shown in this model. It is similar to the SARIMAX study that includes the exogenous variable of the temperature as another input to improve the prediction. However, this scenario is created in order to compare multistep models, to know if more than one exogenous variable can be added and to compare with the statistical multivariate models. In this case, the variables added will be the temperature and the energy price in **Canaria's scenario**. The temperature was taken from the NASA dataset as it was explained in section 5.1, page 27. On the other hand, the other exogenous variable, the energy price, was collected from REE.
The NN selected as the main tool to predict future value with multiple input variables is the **LSTM neural network**.

In this case, the data treatment was different because a new variable was added. All the information was collected and ordered in a dataframe in order to feed the NN as it corresponds. The figure 31 shows the different variables that the NN will manage, being possible to check that the range of time is reduced. In this case, the series goes from January 1st, 2018 to December 31st, 2018. As happened before, it is a daily data series.

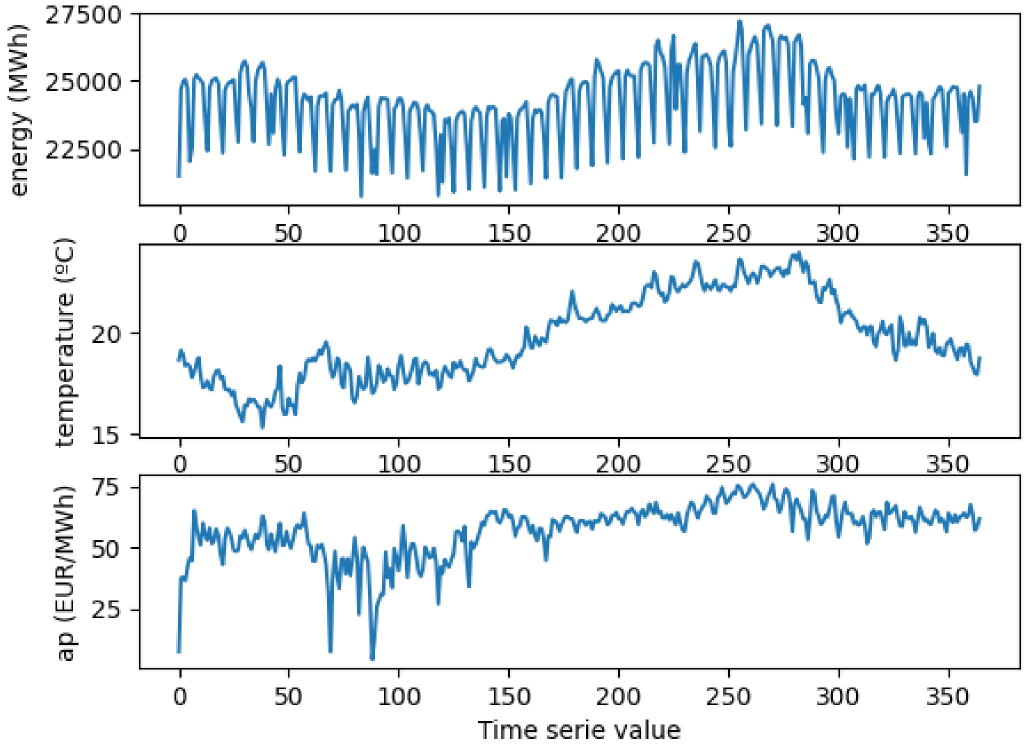


Figure 31: Input variables of an LSTM forecast

This dataframe will feed the model but changing the nature of it in order to suit properly the NN. The code that changes the data series to a supervised series, will add, in this case, the new variables as input to predict future values and, moreover, it will organise the values as it is needed. The part of the code related to do it is the following one[30]:

```python
def series_to_supervised(data, n_in=1, n_out=1, dropnan=True):
  n_vars = 1 if type(data) is list else data.shape[1]
  df = DataFrame(data)
  cols, names = list(), list()
  # input sequence (t-n, ... t-1)
  for i in range(n_in, 0, -1):
        cols.append(df.shift(i))
        names += [('var%d(t-%d)' % (j+1, i)) for j in range(n_vars)]
  # if more than one output is needed
  for i in range(0, n_out):
        cols.append(df.shift(-i))
        if i == 0:
          names += [('var%d(t)' % (j+1)) for j in range(n_vars)]
        else:
          names += [('var%d(t+%d)' % (j+1, i)) for j in range(n_vars)]
  # put it all together
  agg = concat(cols, axis=1)
  agg.columns = names
  if dropnan:
        agg.dropna(inplace=True)
  return agg
```

The output of the function will be a set of data ordered as the LSTM needs. The different var(t-1) represent previous values of the different variables and var1(t) is the energy output. The model was normalized and therefore the range of the model is always lower than 1.

```
   var1(t-1)  var2(t-1)  var3(t-1)   var1(t)
1   0.115848   0.387097   0.043959  0.607237
2   0.607237   0.442396   0.464651  0.655380
3   0.655380   0.415899   0.468291  0.666139
4   0.666139   0.353687   0.448691  0.615000
5   0.615000   0.364055   0.527789  0.198191
```

The model is split into the train and the test part, being the first 265 days, the training series, and the last 100 days, the test series.

Then, the LSTM is developed with **50 training epochs** and a **batch size of 72**. As the others NN, the system is improved paying attention to the mean absolute error but it is scored with the RMSE. The NN seems as follow.

```
model = Sequential()
model.add(LSTM(50, input_shape=(train_X.shape[1], train_X.shape[2])))
model.add(Dense(1))
model.compile(loss='mae', optimizer='adam')
history = model.fit(train_X, train_y, epochs=50, batch_size=72,
  validation_data=(test_X, test_y), verbose=2, shuffle=False)
```

Now, the only missed step is to evaluate and score the model in order to compare with the rest of methods used in this report. This part is carried out by the following part of the code:

```
prov = model.predict(test_X)
test_X = test_X.reshape((test_X.shape[0], test_X.shape[2]))
# invert scaling for forecast
inv_prov = concatenate((prov, test_X[:, 1:]), axis=1)
inv_prov = scaler.inverse_transform(inv_prov)
inv_prov = inv_prov[:,0]
# invert scaling for actual
test_y = test_y.reshape((len(test_y), 1))
inv_y = concatenate((test_y, test_X[:, 1:]), axis=1)
inv_y = scaler.inverse_transform(inv_y)
inv_y = inv_y[:,0]
# calculate RMSE
rmse = sqrt(mean_squared_error(test_y, prov))
print('Test RMSE: %.3f' % rmse)
```

With this configuration, in the Canarias scenario, the result of the RMSE of a multivariate and multistep model was **0,172** which in comparison to the SARIMAX model is not that good, but it is **much faster•** and can take into account more than one exogenous variable easily. The result of the prediction can be seen in figure 32.

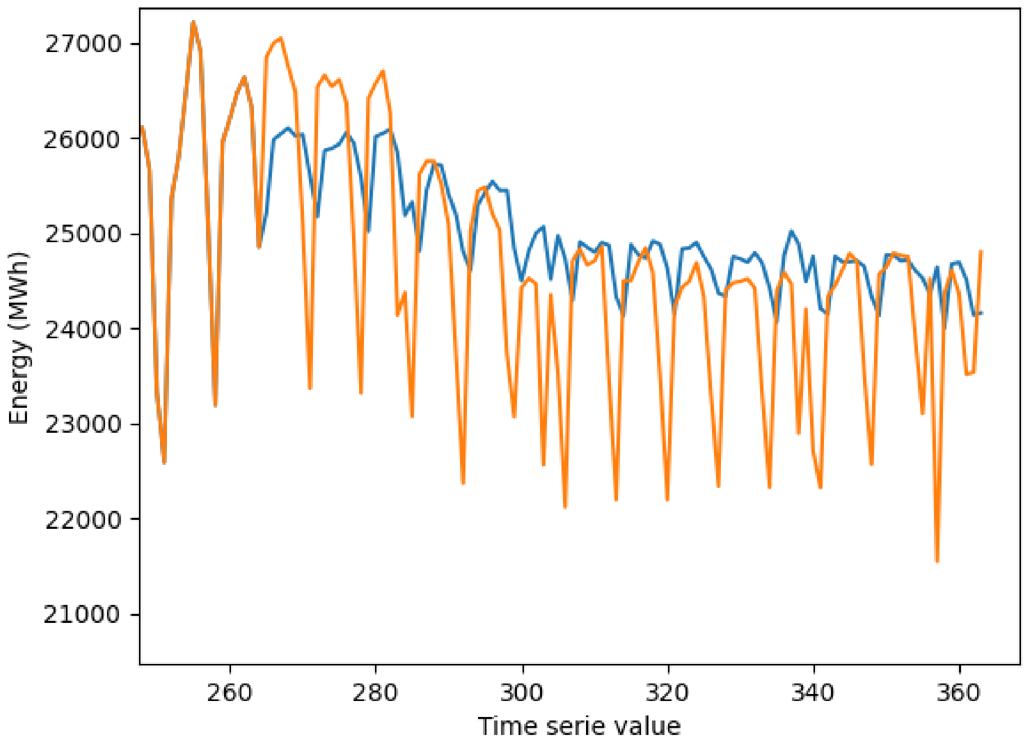


Figure 32: LSTM for multivariate Canarias' scenario

It can not be compared to other scenarios because this time is a **multistep forecast** but it is really close to the actual result and the shape of the output shows the same pattern as the real signal. For being a fast and multistep model, the output given is a good approach even if the RMSE is not close to the other results.

5.3.8 Recurrent Neural Network (RNN) pandemic prediction

This scenario was made in order to check how a prediction method behaves in the case of a strange event such as the covid pandemic of 2020. The scenario chosen was the energy Canary Island energy demand and the model used is the **LSTM-CNN**, which was one of the best models to forecast this scenario.
The advantage of using one of the neural networks is that the results did not vary that much changing the configuration. Therefore, the same configuration as the one used in the RNN for a univariate test (section 5.3.6) will be used.

```
config=[3, 12, 64, 3, 100, 200, 100]
```

The fact of being a univariate model reduces the error because each time that the model makes a prediction is corrected, which must be taken under consideration.
After running the code of the LSTM-CNN network, the result obtained can be seen in figure 33 with a RMSE of 0,049.

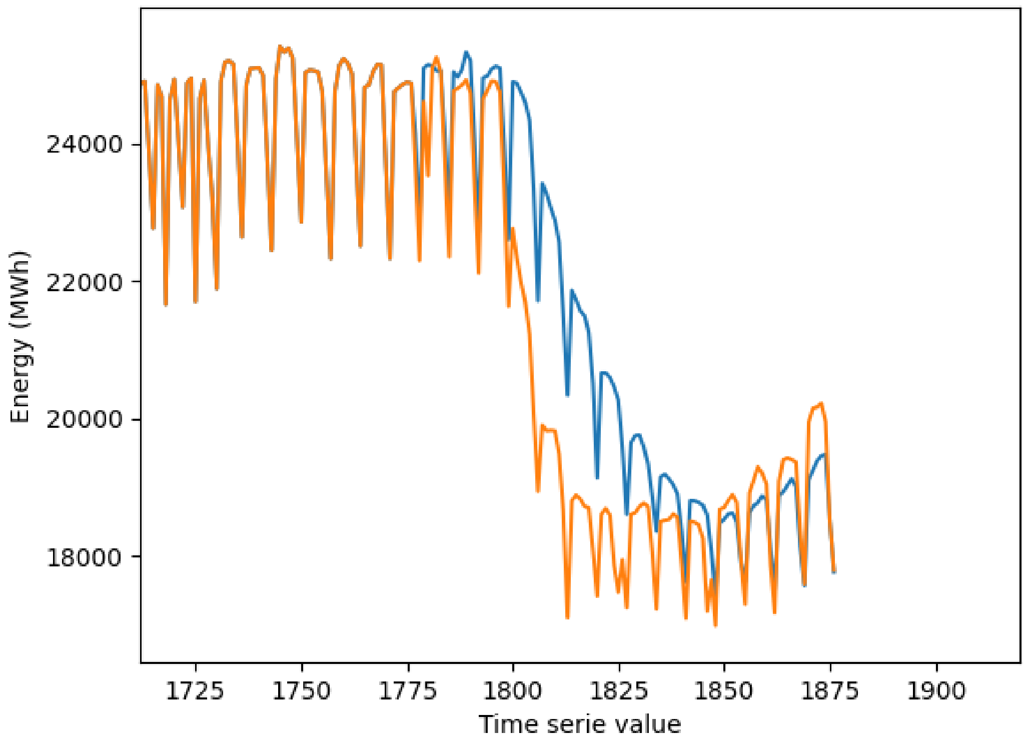


Figure 33: Canarias scenario prediction in during lockdown

Even being a strange event, the results obtained are not really bad because of being a univariate model. However, as the model progresses, it can be seen a delay in the prediction due to the influence of past values. It is for this reason that, to solve this model, it is preferable to be able to forget past events and focus on recent values (fast learning).
As a summary of this scenario, when strange situations that did not happen in the past must be studied, the accuracy of the model is reduced significantly. In the case of statistical methods, this situation can cause noise in the model for future prediction and must be treated to eliminate it. Moreover, with multistep predictions, this impact is much more noticeable.

5.4 Results

The final results are summarized in table 1 and in table 2, the methods are sorted taking into account the average RMSE of both scenarios. The values of the demand prediction scenario using multivariate neural networks and the demand prediction scenario during the pandemic are not included in the results, since they have been carried out only to find out how the models adapt to different situations.

Table 1: RMSE results

RMSE	London	Canarias
Naive-model-persist	0.0794	0.0309
Naive-model-no-persist	0.0957	0.0367
Exponential-smoothing	0.0902	0.0376
SARIMA	0.0833	0.0271
SARIMAX	0.0874	0.0381
CNN	0.0810	0.0280
LSTM-CNN	0.0780	0.0280

Table 2: RMSE best models

Method	RMSE
LSTM-CNN	0.0530
CNN	0.0545
Naive-model-persist	0.0551
SARIMA	0.0552
SARIMAX	0.0627
Exponential-smoothing	0.0639
Naive-model-no-persist	0.0662

Just the LSTM-CNN and the CNN models are the methods that improve the naive model performance and therefore the only valid methods. However, the fact of having an unpredicted historic as happen with the smart counter of only one consumption point must be taken into account and therefore, the results of Canarias scenarios are also sorted in order to classify them taking into account a more stable signal. The results are depicted in table 3.

Table 3: Canarias RMSE best results

RMSE	Canarias
SARIMA	0.0271
CNN	0.0280
LSTM-CNN	0.0280
Naive-model-persist	0.0309
Naive-model-no-persist	0.0367
Exponential-smoothing	0.0376
SARIMAX	0.0381

In the Canarias scenario, the results which improve the performance of the naive models were the SARIMA, the CNN and the LSTM-CNN models. That means that these methods are the most reliable and precise.

However, attention should be paid to another parameter which is the computational cost. That can be measured as the time needed to run the codes but two different tasks must be also separated. The first task is to do a grid search to check which are the best combinations of parameters and the other time-consuming task is the prediction of the last 100 values.
In order to accelerate the process, some models were run in a supercomputer to have the re-

sults faster and two different PC beyond the supercomputers were used to run the codes. The RAM of each supercomputer is depicted in table 4. As the ANN did not have a grid search process, the times shown in the table are those relative to the prediction of the last 100 values. To standardise the results, the times shown are those corresponding to the London scenario.

Table 4: Computational cost

Time to find the best combination			Time to predict 100 values	
Method	Time needed (s)	RAM of the computer (GB)	Time needed (s)	RAM of the computer (GB)
Naive model	7683,22	8	0.050	8
Exponential smoothing	14276,15	8	9.940	8
SARIMA	34290,92	32	8,275.961	16
SARIMAX[3]			9,794.165	16
CNN			536.977	16
LSTM-CNN			536.977	16

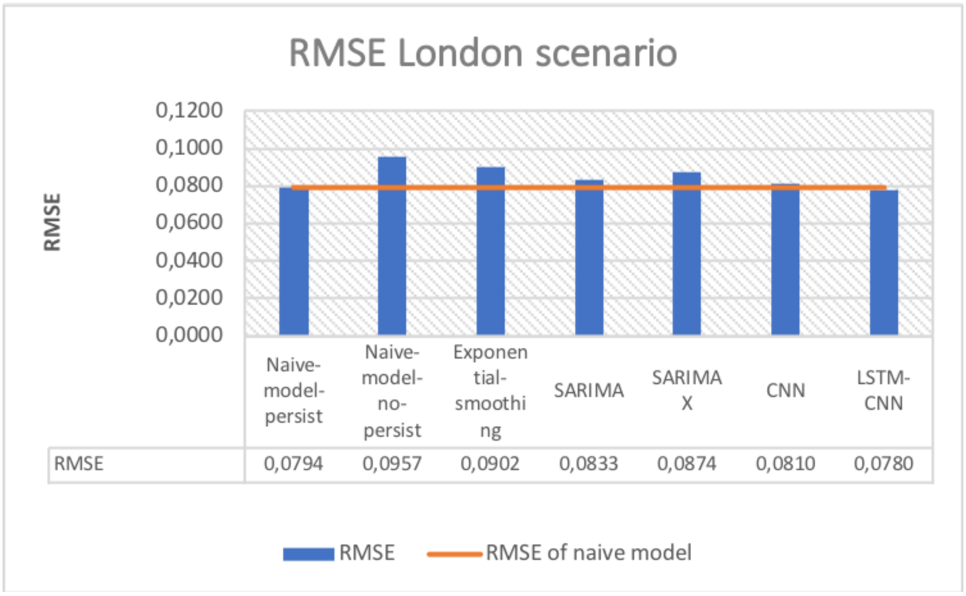
In table 4 can be seen that the fastest model for grid searching was the naive model with a little bit more than two hours followed by the exponential smoothing model which needed approximately two hours more. Because of the computational cost, SARIMA model was run in a supercomputer but nonetheless, the time needed was nine hours and a half with four more RAM capacity than previous models.
On the other hand, ANN needs much more time to do a grid search and therefore, the parameters were set by the user.

In the prediction task, the times were not as big as the grid search analysis which needs to run all the possibilities set before. Therefore the times needed were not long. The naive model and the exponential smoothing were the fastest models followed by the ANN (with similar execution times). However, the SARIMA, SARIMAX and ANN models were run by a better computer in RAM terms.
The models which last around two and three hours were the SARIMA and SARIMAX methods, which, without a doubt, are the most computationally expensive methods.

5.5 Comparison

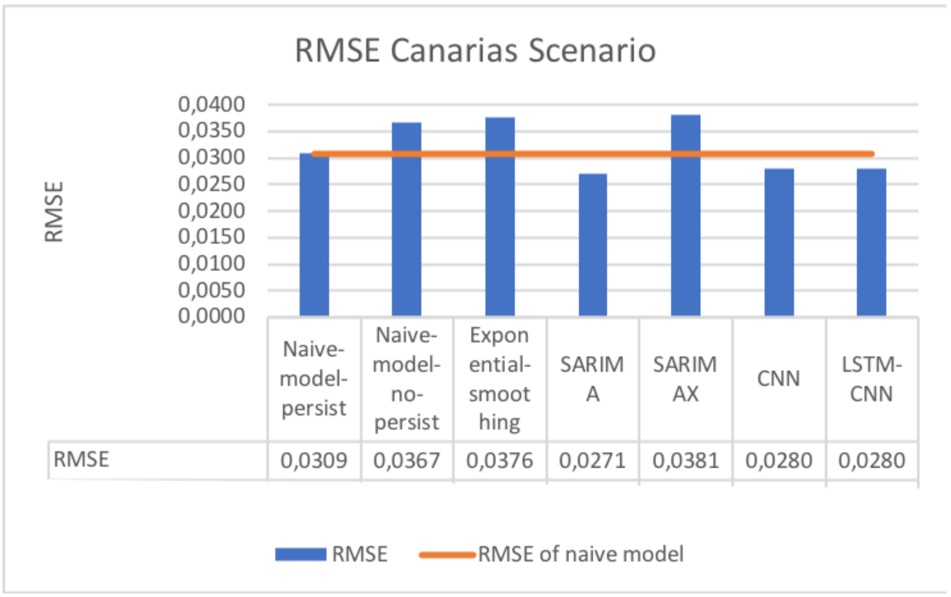
In order to compare the results obtained in the report, the different RMSE errors are depicted in chart 34 and 35.

[3]SARIMAX does not have time to find the best configuration because hte configuration used was exactly the same as the SARIMA model



	Naive-model-persist	Naive-model-no-persist	Exponential-smoothing	SARIMA	SARIMAX	CNN	LSTM-CNN
RMSE	0,0794	0,0957	0,0902	0,0833	0,0874	0,0810	0,0780

Figure 34: RMSE London scenario



	Naive-model-persist	Naive-model-no-persist	Exponential-smoothing	SARIMA	SARIMAX	CNN	LSTM-CNN
RMSE	0,0309	0,0367	0,0376	0,0271	0,0381	0,0280	0,0280

Figure 35: RMSE Canarias scenario

The difference between the naive model (the first model depicted) and the rest of them can show their reliability. Therefore, the RMSE of the naive model is set as the limit that any method can not exceed to use as a prediction method. In the London scenario, the methods that clearly exceed the naive model are the exponential smoothing, the SARIMA and the SARIMAX method. The CNN is close to achieving the same RMSE as the naive model but the fact of not using a grid searching strategy, i.e. guaranteed that uses the best configuration among all the given possibilities, must be taken into account. In this scenario, the best model and the most reliable is the LSTM-CNN.

In the Canarias scenario, which uses more than one smart meter and has a seasonal signal, the methods that overcome the naive model are more. In this scenario, SARIMA, CNN and

LSTM-CNN models overcome the boundaries set to consider reliable models. That means that predictions for scenarios with a larger size, which is the scope of the project, are more reliable.

On the other hand, the existing correlation between temperature and energy is not as strong as one might think at first and, really, it does not have any notable advantages compared to the methods used that do not take this exogenous variable into account. Therefore, can be concluded that the addition of temperature variables does not impact that much to the forecast models.

Furthermore, taking into account the computational cost, ANNs need much more time that statistical methods to do a grid searching process and, therefore, it was skipped in this report. Nevertheless, the prediction process does not need as much time as other methods with similar accuracy beyond the naive model to be executed. Moreover, the different configurations of ANN usually give accurate models regardless of having done or not a grid search before.

Finally, the difference between the accuracy of the models between a stochastic signal as can be the London scenario and a more stable signal as in the Canarias scenario can be seen in figure 36. Being positive values means that the London scenario had a bigger RMSE.

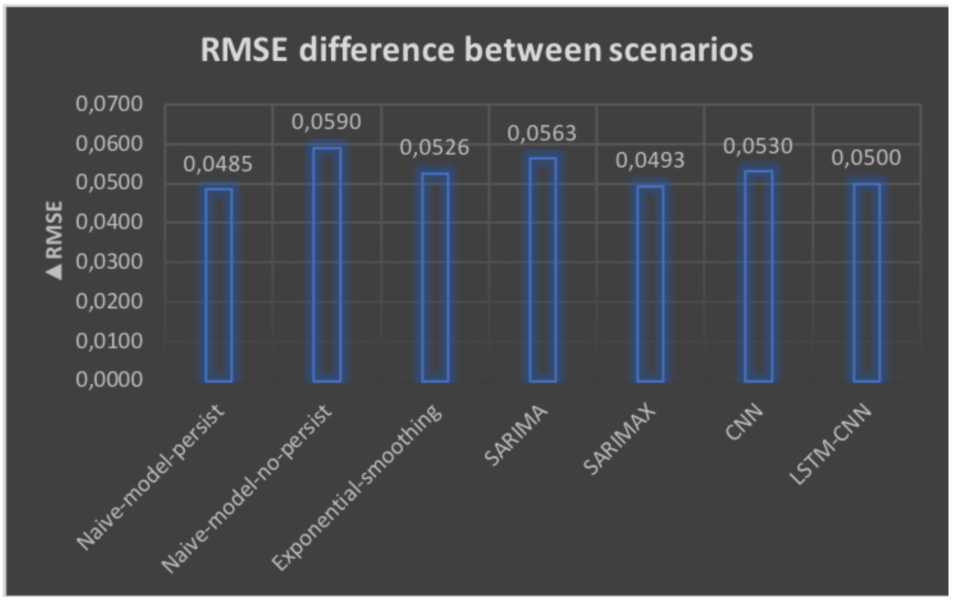


Figure 36: Difference between the RMSE of both scenarios

As can be seen in the figure, all the results show a worse prediction in the London scenario.

Conclusions

This project has analysed statistical and neural network based methods for long term electrical demand forecasting. The models addressed used univariate and multivariate dataset as well as one-step and multi-step forecasting characteristics.
As a first conclusion of this work, as mentioned at the beginning, all the results obtained do not really show which is the best model, but rather the best model for that case at that time. Therefore, one of the advantages of neural networks is that they can give feedback and adjust better compared to statistical methods, thus being more flexible when making predictions in new cases. However, the success of the models really lies in a better understanding of them.

In fact, the results obtained vary quite a bit from the results seen in figure 1, obtained from the book "Predict the Future with MLPs, CNNs and LSTM in Python" by Jason Brownlee. In this last book, it is shown that the exponential smoothing method and the ARIMA method greatly improve ANN models based on their RMSE as well. However, the results obtained in this report do not show the same. It is only SARIMA, the method that coincides as one of the most accurate, but its execution time entails a large computational cost, which compared to the rest of the models does not make it such an attractive model.
Although the objective of the study is the prediction of long-term demand and the time factor is not so crucial, the advantage in terms of accuracy obtained by this model does not make it such a profitable model. Instead, the ANNs have reduced, on average, the RMSE by 9,23% in comparison to statistical methods and they are 88,12% faster than these models.
It is worth mentioning that the variation between the best results after carrying out the grid search process and between the different configurations of the neural networks tried, have not been very noticeable, having the ability to obtain results very close to the best possible in a simpler way.

On the other hand, the models that include exogenous variables have not been more accurate than the models that have not taken them into account. This is due to a relatively not so high real correlation between temperature and energy consumed as shown in the report. In fact, the SARIMAX method has not even overcome the SARIMA method despite entailing greater computational cost. Likewise, the LSTM model with the exogenous variables of temperature and energy cost has given a much worse value than the rest of the models, being only beneficial in terms of the necessary execution time, in which case it improves the SARIMAX method. It is also worth mentioning that the addition of variables in ANN models is simpler than in the case of SARIMAX, whose advantage is clearer when there is more than one exogenous variable.

Finally, it should be noted that the trend tracking of the models has not been similar at all. Although some models have achieved good results, they have not all followed historical trends as well. This is the case of the naive model, the exponential smoothing in the case of the Canary Islands and the multivariate LSTM. In the case of the exponential smoothing method in London scenario, the SARIMA and SARIMAX models and the univariate ANN, the models manage to take into account the variations produced much better. This means that although some models have achieved relatively low errors, their prediction of demand is not always accurate. It would be like considering the average of the history as a reliable tool. Although the final error might be low, its ability to predict trends is not good at all, which is also important in this work.

Acknowledgements

First of all, I would like to thank Mónica Aragüés Pañalba and Antonio Emmanuel Saldaña Gonzalez for guiding me in this final master's thesis and always giving me the best suggestions to move forward with the work. Also, thank them for their patience, dedication and attention that have clearly improved the final result of the work. Likewise, I would like to thank the faculty and my master's classmates for accompanying me during these years and especially, thank Mohammadhossein Tavanaee Shahroodi and Onur Alican for their support during the master.
Finally, I would obviously like to thank my family and friends for always being there.

ETSEIB

Bibliography

[1] Jason Brownlee. A gentle introduction to exponential smoothing for time series forecasting in python. https://machinelearningmastery.com/exponential-smoothing-for-time-series-forecasting-in-python/, 04 2019.

[2] IEEE. Deep in thought. applying artificial intelligence in the grid. *IEEE Power and Energy Magazine*, 20:1–1, 05 2022.

[3] History of Data Science. Dartmouth summer research project: The birth of artificial intelligence. https://www.historyofdatascience.com/dartmouth-summer-research-project-the-birth-of-artificial-intelligence/, 09 2021.

[4] Yan Du, Fangxing Li, Kuldeep Kurte, Jeffrey Munk, and Helia Zandi. Demonstration of intelligent hvac load management with deep reinforcement learning: Real-world experience of machine learning in demand control. *IEEE Power and Energy Magazine*, 20:42–53, 05 2022.

[5] IRENA (International Renewable Energy Agency). *Artifitial Intelligence and Big Data. Innovation Landscape Brief*. IRENA, 2019.

[6] DOCUMENTOS RNE. La historia de la meteorología: los mensajeros del tiempo, 01 2020.

[7] IBM. ¿qué son las redes neuronales?, 08 2020.

[8] Patrycja Mach. 10 business applications of neural network (with examples!), 01 2021.

[9] Andrew Plummer. Box-cox transformation: Explained. https://towardsdatascience.com/box-cox-transformation-explained-51d745e34203, 09 2020.

[10] Zach. Augmented dickey-fuller test in python (with example). https://www.statology.org/dickey-fuller-test-python/, 05 2021.

[11] 365 Data Science. What is an autoregressive model? | 365 datascience. https://365datascience.com/tutorials/time-series-analysis-tutorials/autoregressive-model/, 03 2020.

[12] Jayesh Salvi. Significance of acf and pacf plots in time series analysis. https://towardsdatascience.com/significance-of-acf-and-pacf-plots-in-time-series-analysis-2fa11a5d10a8, 03 2019.

[13] statsmodel. Introduction — statsmodels, 02 2022.

[14] PennState. 14.1 - autoregressive models | stat 501. https://online.stat.psu.edu/stat501/lesson/14/14.1, 2022.

[15] 365 Data Science. What is a moving average model? https://365datascience.com/tutorials/time-series-analysis-tutorials/moving-average-model/, 04 2020.

[16] 365 Data Science. What is an arma model? | 365 data science. https://365datascience.com/tutorials/time-series-analysis-tutorials/arma-model/, 05 2020.

[17] 365 Data Science. What is an arima model? https://365datascience.com/tutorials/python-tutorials/arima/, 06 2020.

[18] Jason Brownlee. *Deep Learning for Time Series Forecasting. Predict the Future with MLPs, CNNs and LSTMs in Python*. Machine Learning Mastery, 2020.

[19] 365 Data Science. What is an arimax model? https://365datascience.com/tutorials/python-tutorials/arimax/, 06 2020.

[20] 365 Data Science. What is a sarimax model? https://365datascience.com/tutorials/python-tutorials/sarimax/, 07 2020.

[21] IBM Cloud Education. What are neural networks? https://www.ibm.com/cloud/learn/neural-networks, 08 2020.

[22] IBM. What are convolutional neural networks? https://www.ibm.com/cloud/learn/convolutional-neural-networks, 08 2020.

[23] IBM Cloud Education. What are recurrent neural networks? https://www.ibm.com/cloud/learn/recurrent-neural-networks, 09 2020.

[24] Alex Graves and Jürgen Schmidhuber. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18:602–610, 07 2005.

[25] Jason Brownlee. A gentle introduction to long short-term memory networks by the experts. https://machinelearningmastery.com/gentle-introduction-long-short-term-memory-networks-experts/, 07 2017.

[26] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 11 1997.

[27] IBM Technology. What is lstm (long short term memory)? https://www.youtube.com/watch?v=b61DPVFX03I, 11 2021.

[28] Jason Brownlee. Cnn long short-term memory networks. https://machinelearningmastery.com/cnn-long-short-term-memory-networks/, 08 2017.

[29] NASA. Nasa power | prediction of worldwide energy resources. https://power.larc.nasa.gov/, 2012.

[30] Jason Brownlee. Multivariate time series forecasting with lstms in keras. https://machinelearningmastery.com/multivariate-time-series-forecasting-lstms-keras/, 09 2018.

[31] Colopremium. Los cartuchos de tinta y su composición - color premium, 01 2022.

A Economic Analysis

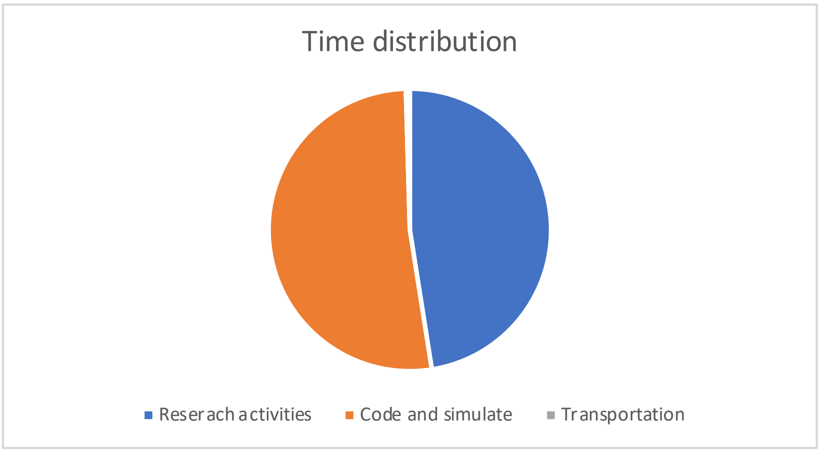
Economic analysis

Billing information	
Billing number	Long-term demand forecast
Reference	00000001
Invoice issue date	44742
Billing period	From 01/02/22 to 30/06/22

Billing and Data Summary	
Power (W)	85
Energy (kWh)	27,37
Discount (5%)	1.110,02 €
Others	0
Taxes	5359,065558

Elec. Cons. (kWh)	Price ele. (EUR/kWh)	Total (EUR)
27,37	0,148	4,05 €

Activities	Number of hours (h)	Price (EUR/h)	Total (EUR)
Reserach activities	420	20,00 €	8.400,00 €
Code and simulate	460	30,00 €	13.800,00 €
Transportation	4	0,10 €	0,40 €
Discount (5%)			1.110,02 €
IVA			4.428,98 €
Total	**884**		**25.519,36 €**



The information added in the economic analysis is mainly related to the time spent doing the report. In total, the time spent was **884 hours** for, basically, **research and coding tasks**. Writing the document is included in the research item.
The energy depicted in the economic terms is the computer's consumption which is the unique tool needed for doing the research. It is multiplied by a coefficient of 0,7 because it used also energy stored in the battery.
The final cost is the sum of all costs plus the taxes (IVA 21%). The final cost, as can be seen in the bill is **25.519,36 €**.

B Environmental Impact

In this Annex, the environmental impact of the report will be analysed and interpreted. The tool used to measure the impact is the **Life Cycle Assessment (LCA)** that was done with the openLCA© program.

B.1 Goal and scope

The element studied is the **creation of a paper** taking into account all the simulations that have been carried out and the **computational cost** of the project. Moreover, the **print process** to have a paper version is also added in this analysis. For this, a life cycle assessment will be carried out.
The aim of the LCA is to have general information on the environmental impact that the creation of a paper has. The reason for carrying out this assessment is for academic purposes and to deliver it to the ETSEIB faculty the results.
As it was said above, openLCA was the program used to do the analysis and the database selected was an *ecoinvent database*. The assessment done is a **cradle to grave analysis** where the different stages of the project were taken into account. Furthermore, there are some cut-off rules. For instance, the elements which have a **lower than the 1%** of impact were **neglected**[4]. Also, the providers selected in the database have cut-off rules in order to simplify and generalised the results. The unit of analysis was a final report.

B.2 Inventory and Analysis

The model will consist, in the first place, of the energy consumed by the computer, which, in order to simplify the analysis, it will be considered that the study was carried out by a single laptop whose energy consumption data is reflected in Annex A. And, second, on the ink and paper consumed to print the work. For the ink, some components have been omitted due to low impact and the following composition has been considered, assuming the consumption of a quarter of an ink cartridge[31].

Table 5: Paper ink composition

12% diethylene	0,705 g
81% heavy water	0,47 g

The final amounts were calculated thanks to the total weight consumed by printing the document which was set as $5,875\ g$.
The scheme of the process that was taken into account in the analysis can be seen in figure 37.

[4]To do this, a first draft was done in order to classified the impacts.

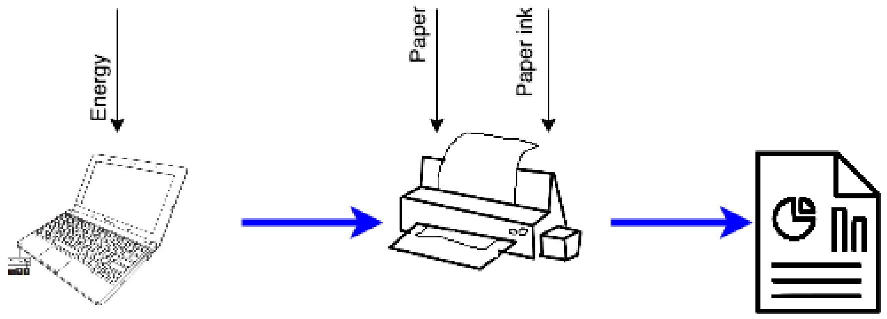


Figure 37: Report flow

To quantify the model, different assumptions were done which were needed because the lack of data. In table 6, the different assumptions are depicted.

Table 6: Assumptions

Assumptions	
Pages of the report	90 pages
Weigth of each page	20 g
Total weight	1800 g
Volume of a paper ink cartridge	23,5 cm^3
density of a high water content paper ink	1 g/cm^3
Weight of a paper ink cartridge	23,5 g
Volume used to print the report	1/4 of total = 5,875 g

Being the final quantities the ones depicted in table 7.

Table 7: Inventory of openLCA

Inventory	
Diethylene glycol	0.705 g
Heavy water	0.47 g
Paper	1800 g
Electricity	27.37 kWh

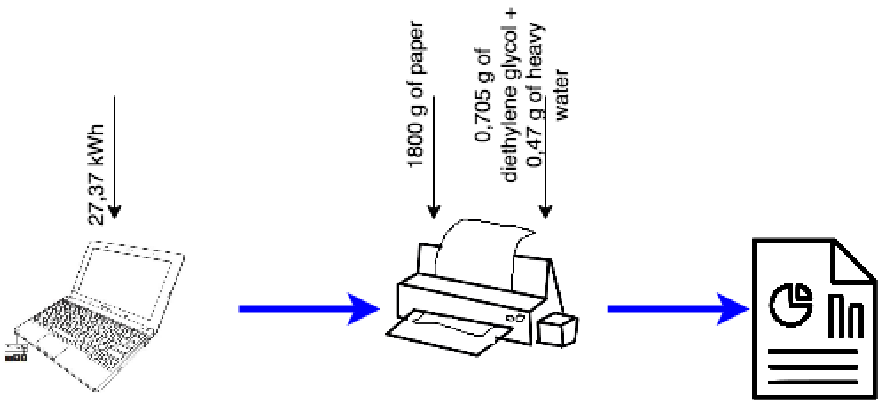


Figure 38: Report flow

B.3 Impact Analysis

From the results obtained, **three of them were highlighted**. The first one is the **aquatic ecotoxicity** produced by doing the report. The units are **kg TEG water** and the biggest impact is related to the electricity consumed. Obviously, because of the size, all the results depicted a larger impact due to the electricity consumption, which could be expected. For heavy water and paper production, a small impact, in comparison to electricity consumption, can be seen.

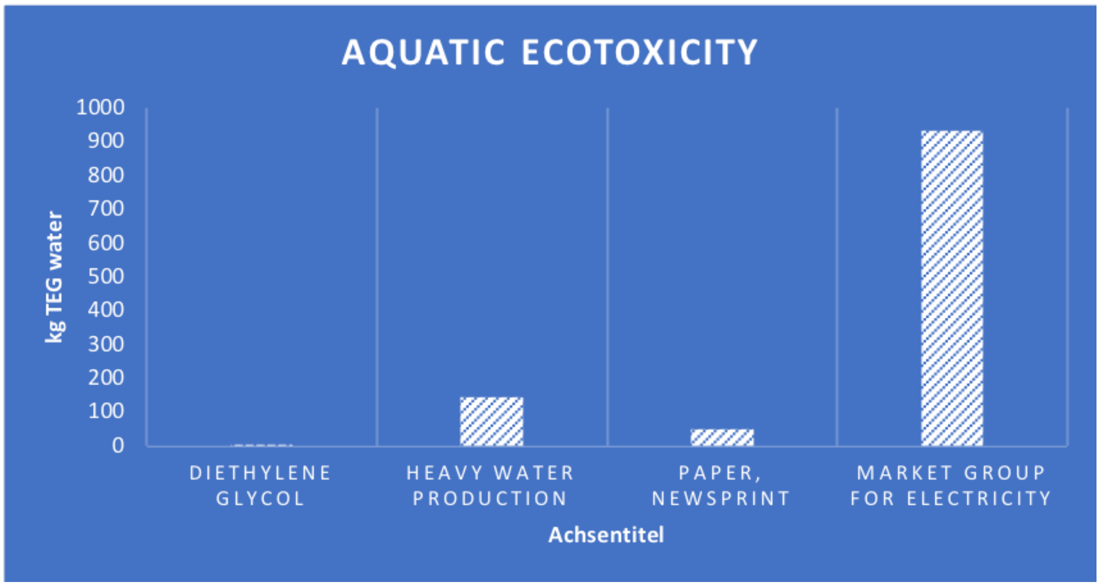


Figure 39: Aquatic ecotoxicity

Second important parameter is the **Global Warming** measured in $kg\ CO_{2\ eq}$. In this case, electricity consumption stands out from the rest with $10,76\ kg\ CO_{2\ eq}$, being the sum of the impacts $10,95\ kg\ CO_{2\ eq}$. The results are shown in figure 40

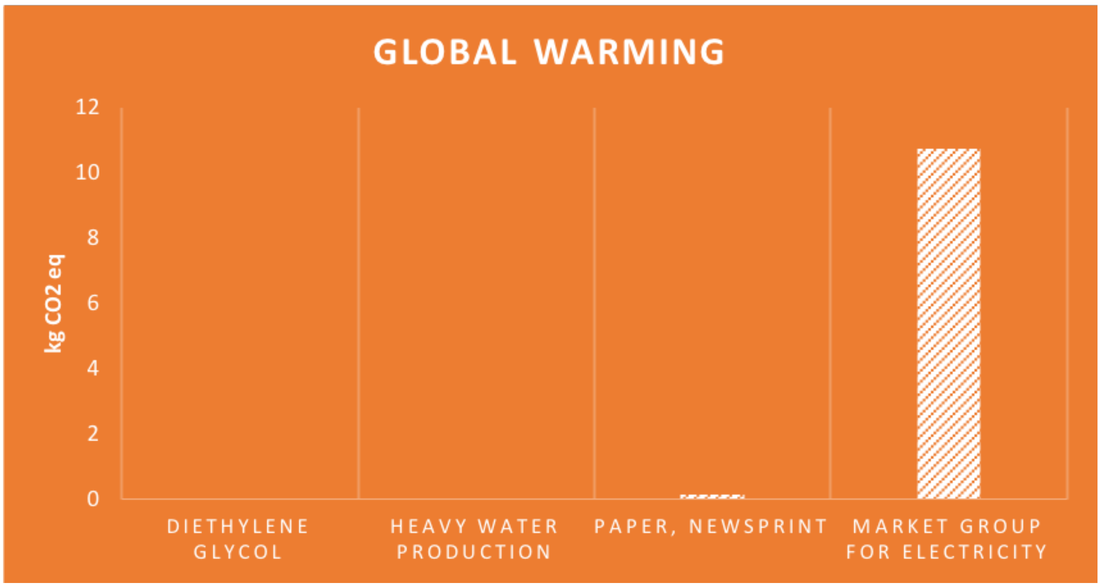


Figure 40: Global Warming Effect

Last parameter added in the analysis is the **non-renewable energy consumed** which, as happened before, electricity consumption is the responsible for most of the impact in this group. The units of this category are MJ and the contribution of electricity consumption was $241,39\ MJ$ out of $244,58\ MJ$.

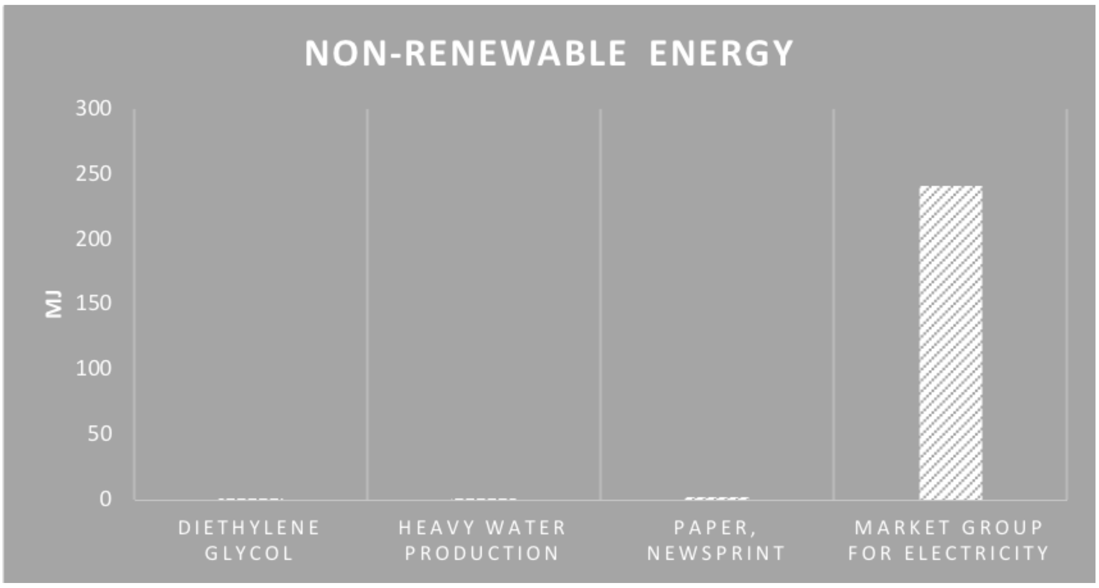


Figure 41: Non-renewable energy

B.4 Interpretation

In the report, there are some **uncertainties** such as, where the papers source or the fact of neglecting the electricity consumption of all internet databases consulted. Moreover, the paper

ink used was an estimation and is not really accurate. All these uncertainties must be taken into account in order to interpret the results.
The impact of this report is hard to calculate. In comparison to fabrication processes, the results obtained are not very important because research processes do not impact that much. As can be seen in the analysis, the largest impacts are due to electricity consumption.
Because of the increment of electricity usage, most of the information is managed and shown through the internet and, therefore, all the impact products are being summarised in one: electricity. There is a reduction in the environmental impact due to products such as paper because there is not as common to deliver all the reports in paper versions as before but through the internet.
Summing up, the largest impact can be expected as the electricity consumption because most of the research was done through the computer.

C Data treatment Code

```
#--------DATASET WITH ALL THE INFORMATION--------
#IMPORT THE USEFUL LIBRARIES
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

#Import the relevant .csv files
dftotal1 = pd.read_csv("block_0.csv",)
#Import the information related with the energy values
dfweather = pd.read_csv('powerdata.csv')
#Import the information related with the weather values

#Filtering to the values of one id
mac=dftotal1[dftotal1['LCLid']=='MAC000002'] #Select just one id

#--------WEATHER--------
#Change the variable names to use to_datetimecommand() correctly
dfweather.columns=['year', 'month', 'day', 'hour',
        'temperature (C)', 'precipitation']
dfweathertime=pd.DataFrame(pd.concat([dfweather['year'],
        dfweather['month'], dfweather['day'], dfweather['hour']],axis=1))
#Separate those related with time
#Change the datetime format
dfweathertime=pd.to_datetime(dfweathertime)
dfweathertime.name='time'
#Join time to weather dataframe
dfweather=pd.merge(dfweather, dfweathertime, how='outer', left_index=True,
right_index=True)
dfweather=dfweather.set_index(['time'])

#--------ENERGY--------
#Change the energy column to numeric values and use NaN values in those
#with error in them. Then, remove NaN values.
mac['energy(kWh/hh)']=pd.to_numeric(mac['energy(kWh/hh)'], errors='coerce')
#it changes the type of the column energy to numeric values and
#if any error, the answer will be NaN values

#This loop is done use hourly energy instead of half and hour energy information
first_value=mac.iloc[0]['energy(kWh/hh)']
variable=[]
index=[]
for i in range(0,len(mac)):
    variable.append(mac.iloc[i]['energy(kWh/hh)']
                +mac.iloc[i-1]['energy(kWh/hh)'])
variable[0]=first_value #to don't affect the first value and use the value 0.0
index=mac['tstp']
#Create the dataframe with the variables that we want
#Convert first to serie and then to a frame the index
serie_index=pd.Series(index)
serie_index = serie_index.to_frame(name='index')
#Convert first to serie and then to a frame the energy
serie_variable=pd.Series(variable)
```

```
serie_variable = serie_variable.to_frame(name='energy (kWh)')
#Join both datasets
dfenergy=pd.concat([serie_index, serie_variable], axis=1)
#Create a dataframe
dfenergy=pd.DataFrame(dfenergy)
dfenergy['index']=pd.to_datetime(dfenergy['index']) #Change the datetime format
dfenergy=dfenergy.set_index(dfenergy['index']) #Set the index

#--------COMBINATION OF DATAFRAMES--------
# Merge the dataframes with the useful information
dftotal=pd.merge(dfenergy['energy (kWh)'], dfweather['temperature (C)'],
        how='outer', right_index=True, left_index=True)
dftotal=dftotal.dropna() #Elminate nan answers

dftotal.to_csv("final_dataframe.csv")
```

D Naive Code

```
######NAIVE MODEL######
#LIBRARIES
from pandas import read_csv
import pandas as pd
from matplotlib import pyplot
from math import sqrt
import numpy as np
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from sklearn.metrics import mean_squared_error
import time
startTime = time.time()

def simple_forecast(history, configuration):
  #the configuration is the n last values used, the offset and the type
  n, offset, prediction_method = configuration
  # if persist value, ignore other configuration
  if prediction_method == 'persist':
        return history[-n] #use last values
  # collect values to average
  values = list()
  #organise the data depending on the offset component
  if offset == 1:
        values = history[-n:]
  else:
        # skip bad configurations
        if n*offset > len(history):
          raise Exception('The configurationuration is beyond end
                of data: %d * %d > len(data). It is not posible to do
                the forecast' % (n,offset))
        # try and collect n values using offset
        for j in range(1, n+1):
          m = j * offset
          values.append(history[-m])
  # check if we can average
  if len(values) < 2:
        raise Exception('Imposible to calculate average')
  # mean of last n values
  if prediction_method == 'mean':
        return np.mean(values)
  # median of last n values
  return np.median(values) #Is better for non-Gaussian distribution

#CHECK THE ERROR PRODUCED IN THE TEST DATASET WITH THE RMSE
def measure_rmse(actual, prediction):
        return sqrt(mean_squared_error(actual, prediction))

#FIT THE MODEL DIVIDING THE DATASET IN TRAIN AND TEST SETS
def split_dataset(data, num_test):
        return data[:-num_test], data[-num_test:]

def walk_forward_validation(data, num_test, config):
  global predictions
  predictions = list()
  # split dataset
  train, test = split_dataset(data, num_test)
  # feed history with training dataset
  history = [x for x in train]
  for i in range(len(test)):
        # fit model and make forecast for history
        prov = simple_forecast(history, config)
        # store forecast in list of predictions
        predictions.append(prov)
        # add actual observation to history for the next loop
        history.append(test[i])
  # estimate the error
  error = measure_rmse(test, predictions)
  return error

def score_model(data, num_test, config, debug=False):
  result = None
  # convert configuration to a string
  string = str(config)
  # show all warnings and fail on exception if debugging
  if debug:
        result = walk_forward_validation(data, num_test, config)
  else:
        # one failure during model validation means an unstable conf.
        try:
          # never show warnings when grid searching because it is noisy
          with catch_warnings():
                filterwarnings("ignore")
                result = walk_forward_validation(data, num_test, config)
        except:
          error = None
  # check for an interesting result
  if result is not None:
        print(' > Model[%s] %.3f' % (string, result))
  return (string, result)

def grid_search(data, config_list, num_test, parallel=True):
  scores=None
  if parallel:
        # executor is used to speed up the grid searching after
        # evaluating the number of cores of the computer
        executor=Parallel(n_jobs=cpu_count(), backend='multiprocessing')
        # define tasks (score_model() function for each configuration)
        tasks = (delayed(score_model)
          (data, num_test, config) for config in config_list)
        # execute list of tasks
        scores = executor(tasks)
  else:
    # execute tasks sequentially
        scores = [score_model
```

```
          (data, num_test, config) for config in config_list]
  #remove empty results
  scores = [r for r in scores if r[1]!=None]
  #sort configurations by error, asc
  scores.sort(string=lambda tup: tup[1])
  return scores

def simple_configurations(max_len, offsets=[1]):
  configurations=list()
  for i in range(1, max_len+1):
        for o in offsets:
          for t in ['persist', 'mean', 'median']:
                config=[i, o, t]
                  configurations.append(config)
  return configurations

if __name__ == '__main__':
  series = read_csv('final_dataframe.csv', header=0, index_col=0)
  del series['temperature (C)']
  val_max=series.max()
  series=series/val_max
  data=series.values
  # to split the dataset
  num_test = 100
  max_len = len(data) - num_test
  config_list = simple_configurations(max_len)
  scores = grid_search(data, config_list, num_test)
  # top 3 configurations
  for config, error in scores[:3]:
        print(config, error)
  # Plot best result
  config=[1175, 1, 'persist']
  series=series*val_max
  data=series.values
  walk_forward_validation(data, num_test, config)
  prediction_values=list()
  for i in range(max_len):
        prediction_values.append(data[i])
  for i in range(num_test):
        prediction_values.append(predictions[i])
  pyplot.plot(prediction_values)
  pyplot.plot(data)
  pyplot.ylabel('Energy (kWh)')
  pyplot.xlabel('Time serie value')
  pyplot.show()
```

E Exponential Smoothing Code

```
#######EXPONENTIAL SMOOTHING CODE#######
#Libraries for Exponential Smoothing
from math import sqrt
from matplotlib import pyplot
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from sklearn.metrics import mean_squared_error
from pandas import read_csv
from numpy import array
import time
startTime = time.time()

def exp_smoothing_forecast(history, configuration):
  t,d,s,p,bo,b = configuration
  history = array(history)
  model = ExponentialSmoothing(history, trend=t, damped_trend=d,
    seasonal=s, seasonal_periods=p, use_boxcox=bo)
  model_fit = model.fit(optimized=True, remove_bias=r)
  prov = model_fit.predict(len(history), len(history))
  return prov[0]

# RMSE
def measure_rmse(actual, prediction):
  return sqrt(mean_squared_error(actual, prediction))

# split dataset
def split_dataset(data, num_test):
  return data[:-num_test], data[-num_test:]

# walk-forward validation for dataset
def walk_forward_validation(data, num_test, config):
  global predictions
  predictions = list()
  # to split dataset
  train, test = split_dataset(data, num_test)
  # feed history with training dataset
  history = [x for x in train]
  for i in range(len(test)):
        prov = exp_smoothing_forecast(history, config)
        predictions.append(prov)
        history.append(test[i])
  # estimate the error
  error = measure_rmse(test, predictions)
  return error

# score a model and return None on failure
def score_model(data, num_test, config, debug=False):
  result = None
  # convert configuration to a string
  string = str(config)
  if debug:
        result = walk_forward_validation(data, num_test, config)
  else:
        try:
          with catch_warnings():
          filterwarnings("ignore")
          result = walk_forward_validation(data, num_test, config)
        except:
          error = None
  # check for an interesting result
  if result is not None:
        print(' > Model[%s] %.3f' % (string, result))
  return (string, result)

def grid_search(data, config_list, num_test, parallel=True):
  scores = None
  if parallel:
        executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
        tasks = (delayed(score_model)(data, num_test, config) for config in config_lis
        scores = executor(tasks)
  else:
        scores = [score_model(data, num_test, config) for config in config_list]
  scores = [r for r in scores if r[1] != None]
  scores.sort(string=lambda tup: tup[1])
  return scores

# create a set of exponential smoothing configurations to try
def exp_smoothing_configurations(seasonal=[None]):
  models = list()
  # define the configuration lists
  trend_p = ['add', 'mul', None]
  damping_p = [True, False]
  seasonal_p = ['add', 'mul', None]
  # seasonal=[None] by default
  # perior_p must integrate the seasonal parameter of the model
  perior_p = seasonal
  boxcox_p = [True, False]
  bias_p = [True, False]
  # create configuration instances
  for t in trend_p:
        for d in damping_p:
          for s in seasonal_p:
                for p in perior_p:
                  for bo in boxcox_p:
                    for b in bias_p:
                          config = [t,d,s,p,bo,b]
                          models.append(config)
  return models
if __name__ == '__main__':
  series = read_csv('final_dataframe.csv', header=0, index_col=0)
  del series['temperature (C)']
  val_max=series.max()
  series=series/val_max
```

```
data = series.values
# split dataset
num_test = 100
max_len = len(data) - num_test
# model configurations without seasonal as default
config_list = exp_smoothing_configurations()
scores = grid_search(data[:,0], config_list, num_test)
# top 3 configurations
for config, error in scores[:3]:
	print(config, error)
# to know the time needed
executionTime = (time.time() - startTime)
print('Execution time in seconds: ' + str(executionTime))
#Plot best result
config=[None, False, None, None, False, False]
series=series*val_max
data=series.values
walk_forward_validation(data, num_test, config)
prediction_values=list()
for i in range(max_len):
	prediction_values.append(data[i])
for i in range(num_test):
	prediction_values.append(predictions[i])
pyplot.plot(prediction_values)
pyplot.plot(data)
pyplot.ylabel('Energy (kWh)')
pyplot.xlabel('Time serie value')
pyplot.show()
```

F SARIMA Code

```
#######SARIMA#######
#Import Libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.stattools import adfuller
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from math import sqrt
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from statsmodels.tsa.statespace.sarimax import SARIMAX
from sklearn.metrics import mean_squared_error

def sarima_forecast(history, config):
    order, sorder, trend = config
    model = SARIMAX(history, order=order, seasonal_order=sorder, trend=trend,
                enforce_stationarity=False, enforce_invertibility=False)
    model_fit = model.fit(disp=False)
    # make one step forecast
    yhat = model_fit.predict(len(history), len(history))
    return yhat[0]

# root mean squared error or rmse
def measure_rmse(actual, predicted):
    return sqrt(mean_squared_error(actual, predicted))
# split into train and test sets
def train_test_split(data, n_test):
    return data[:-n_test], data[-n_test:]

def walk_forward_validation(data, n_test, cfg):
    global predictions
    predictions = list()
    train, test = train_test_split(data, n_test)
    # feed history
    history = [x for x in train]
    for i in range(len(test)):
        yhat = sarima_forecast(history, cfg)
        # save prediction
        predictions.append(yhat)
        # add observation to history
        history.append(test[i])
    # estimate prediction error
    error = measure_rmse(test, predictions)
    return error

# score a model
def score_model(data, n_test, cfg, debug=False):
    result = None
    key = str(cfg)
    if debug:
        result = walk_forward_validation(data, n_test, cfg)
    else:
        try:
            with catch_warnings():
                filterwarnings("ignore")
                result = walk_forward_validation(data, n_test, cfg)
        except:
            error = None
    if result is not None:
        print(' > Model[%s] %.3f' % (key, result))
    return (key, result)

def grid_search(data, cfg_list, n_test, parallel=True):
    scores = None
    if parallel:
        executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
        tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
        scores = executor(tasks)
    else:
        scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
    # remove empty results
    scores = [r for r in scores if r[1] != None]
    # sort configurations by error
    scores.sort(key=lambda tup: tup[1])
    return scores

def sarima_configs(seasonal=[0]):
    models = list()
    p_params = [0, 1, 2, 3 ,4 ,5 ,6 ,7]
    d_params = [0, 1]
    q_params = [0, 1, 2, 3, 4, 5, 6, 7]
    t_params = ['n','c','t','ct']
    P_params = [0, 1, 2]
    D_params = [0, 1]
    Q_params = [0, 1, 2]
    m_params = seasonal
    # create config instances
    for p in p_params:
        for d in d_params:
            for q in q_params:
                for t in t_params:
                    for P in P_params:
                        for D in D_params:
                            for Q in Q_params:
                                for m in m_params:
                                    cfg = [(p,d,q), (P,D,Q,m), t]
                                    models.append(cfg)
    return models


if __name__ == '__main__':
    # define dataset
    series = pd.read_csv('Energia_Canarias/energiacanarias20152019.csv',
                sep=';', header=0, index_col=0)
```

```
val_max=series.max()
series=series/val_max
data = series.values
# data split
n_test = 100
max_length = len(data) - n_test
# model configs
cfg_list = sarima_configs()
# grid search
scores = grid_search(data, cfg_list, n_test)
print('done')
# list top 3 configs
for cfg, error in scores[:3]:
   print(cfg, error)

dta=series
plot_acf(dta.values.squeeze(), lags=n_test, zero=False);
plt.show()
plot_pacf(dta.values.squeeze(), lags=n_test, zero=False, method='ywm');
plt.show()
```

G SARIMAX Code

```
######SARIMAX######
######LIBRARIES######
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
#import seaborn as sns
from statsmodels.tsa.stattools import adfuller #para *1
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from math import sqrt
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from statsmodels.tsa.statespace.sarimax import SARIMAX
from sklearn.metrics import mean_squared_error
import time
startTime = time.time()

def sarimax_forecast(history, temp_history, configuration):
  order, sorder, trend = configuration
  model = SARIMAX(history, exog=temp_history, order=order,
    seasonal_order=sorder, trend=trend, enforce_stationarity=False,
    enforce_invertibility=False)
  model_fit = model.fit(disp=False)
  prov = model_fit.predict(len(history), len(history), exog=temp_history[-1])
  return prov[0]

# RMSE
def measure_rmse(actual, prediction):
  return sqrt(mean_squared_error(actual, prediction))

def split_dataset(data, num_test):
  return data[:-num_test], data[-num_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, num_test, config):
  global predictions
  predictions = list()
  # split the dataset
  train, test = split_dataset(data, num_test)
  train_t, test_t = split_dataset(temp, num_test)
  history = [x for x in train]
  temp_history = [x for x in train_t]
  for i in range(len(test)):
    prov = sarimax_forecast(history, temp_history, config)
    predictions.append(prov)
    history.append(test[i])
    temp_history.append(test_t[i])
  # estimate the error
  error = measure_rmse(test, predictions)
  return error
```

```
if __name__ == '__main__':
  series = pd.read_csv('final_dataframe.csv', header=0, index_col=0)
  del series['energy (kWh)']
  temp = series.values
  series = pd.read_csv('final_dataframe.csv', header=0, index_col=0)
  del series['temperature (C)']
  val_max=series.max()
  series=series/val_max
  data = series.values
  num_test = 100
  max_len = len(data) - num_test

  executionTime = (time.time() - startTime)
  print('Execution time in seconds: ' + str(executionTime))

  #Plot best result
  config=[(6, 0, 6), (0, 0, 0, 0), 'c']
  series=series*val_max
  data=series.values
  result=walk_forward_validation(data, num_test, config)
  prediction_values=list()
  for i in range(max_len):
    prediction_values.append(data[i])
  for i in range(num_test):
    prediction_values.append(predictions[i])
  plt.plot(prediction_values)
  plt.plot(data)
  plt.plot(temp)
  plt.ylabel('Energy (kWh)')
  plt.xlabel('Time serie value')
  plt.show()
  fig, axs = plt.subplots(2)
  axs[0].plot(prediction_values)
  axs[0].plot(data)
  axs[1].plot(temp, color='green')
  for ax in axs.flat:
    ax.set(xlabel='Time serie value', ylabel='Energy (kWh)')
  plt.show()
```

H CNN Code

```
#######CNN model#######
from math import sqrt
from numpy import array
from numpy import mean
from numpy import std
from pandas import DataFrame
from pandas import concat
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from matplotlib import pyplot
import time
startTime = time.time()

# split dataset
def split_dataset(data, num_test):
  return data[:-num_test], data[-num_test:]

# transform list into supervised learning format
def series_to_supervised(data, n_in, n_out=1):
  df = DataFrame(data)
  cols = list()
  # input sequence (t-n, ... t-1)
  for i in range(n_in, 0, -1):
        cols.append(df.shift(i))
  # if it is more than one output...
  for i in range(0, n_out):
        cols.append(df.shift(-i))
  # put it all together
  agg = concat(cols, axis=1)
  # drop rows with NaN values
  agg.dropna(inplace=True)
  return agg.values

# root RMSE
def measure_rmse(actual, prediction):
  return sqrt(mean_squared_error(actual, prediction))

def model_fit(train, configuration):
  n_input, n_filters, n_kernel, n_epochs, n_batch = configuration
  # prepare data
  data = series_to_supervised(train, n_input)
  train_x, train_y = data[:, :-1], data[:, -1]
  #reshape the model
  train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], 1))
  # The different layers of the CNN models are:
  model = Sequential()
  model.add(Conv1D(n_filters, n_kernel, activation='relu',
    input_shape=(n_input, 1)))
  model.add(Conv1D(n_filters, n_kernel, activation='relu'))
  model.add(MaxPooling1D())
  model.add(Flatten())
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch,
    verbose=0)
  return model

def model_predict(model, history, configuration):
  n_input, _, _, _, _ = configuration
  # conver to from 2D to 3D data
  x_input = array(history[-n_input:]).reshape((1, n_input, 1))
  # forecast
  prov = model.predict(x_input, verbose=0)
  return prov[0]

# split the sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
        # find the end of this pattern
        end_ix = i + n_steps
        # check if the value is beyond the sequence
        if end_ix > len(sequence)-1:
      break
        # gather input and output parts of the pattern
        seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
        X.append(seq_x)
        y.append(seq_y)
  return array(X), array(y)


# walk-forward validation for univariate data
def walk_forward_validation(data, num_test, config):
  global predictions
  predictions = list()
  train, test = split_dataset(data, num_test)
  model = model_fit(train, config)
  history = [x for x in train]
  for i in range(len(test)):
        prov = model_predict(model, history, config)
        predictions.append(prov)
        history.append(test[i])
  # estimate the error
  error = measure_rmse(test, predictions)
  print(' > %.3f' % error)
  return error

# repeat 30 times to reduce stoch
def repeat_evaluate(data, configuration, num_test, n_repeats=30):
  best = [walk_forward_validation(data, num_test, configuration)
    for _ in range(n_repeats)]
  return best

# summarize the model
def summarize_best(name, best):
  best_m, score_std = mean(best), std(best)
  print('%s: %.3f RMSE (+/- %.3f)' % (name, best_m, score_std))

####Configuration###
series = read_csv('final_dataframe.csv', header=0, index_col=0)
del series['temperature (C)']
val_max=series.max()
series=series/val_max
data = series.values
# to split dataset
num_test = 100
configuration = [100, 100, 5, 100, 100]
# grid search
best = repeat_evaluate(data, configuration, num_test)
# score
summarize_best('cnn', best)

executionTime = (time.time() - startTime)
print('Execution time in seconds: ' + str(executionTime))

data = series.values
# data split
num_test = 100
# define configuration
configuration = [24, 100, 4, 100, 100]
# grid search
best = walk_forward_validation(data, num_test, configuration)
# summarize best
summarize_best('cnn', best)
series=series*val_max
data=series.values
result=walk_forward_validation(data, num_test, configuration)
prediction_values=list()
for i in range(len(data)-num_test):
  prediction_values.append(data[i])
for i in range(num_test):
  prediction_values.append(predictions[i])
pyplot.plot(prediction_values)
pyplot.plot(data)
pyplot.ylabel('Energy (MWh)')
pyplot.xlabel('Time serie value')
pyplot.show()
```

I CNN-LSTM Code

```
#######CNN-LSTM#######
from math import sqrt
from numpy import array
from numpy import mean
from numpy import std
from pandas import DataFrame
from pandas import concat
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import TimeDistributed
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from matplotlib import pyplot
import time
startTime = time.time()

def split_dataset(data, num_test):
  return data[:-num_test], data[-num_test:]

def series_to_supervised(data, n_in, n_out=1):
  df = DataFrame(data)
  cols = list()
  # input sequence (t-n, ... t-1)
  for i in range(n_in, 0, -1):
        cols.append(df.shift(i))
  # if more than one output is needed...
  for i in range(0, n_out):
        cols.append(df.shift(-i))
  # put it all together
  agg = concat(cols, axis=1)
  # drop rows with NaN values
  agg.dropna(inplace=True)
  return agg.values

# RMSE
def measure_rmse(actual, prediction):
  return sqrt(mean_squared_error(actual, prediction))

def model_fit(train, configuration):
  # unpack configuration
  n_seq, n_steps, n_filters, n_kernel, n_nodes, n_epochs, n_batch = configuration
  n_input = n_seq * n_steps
  data = series_to_supervised(train, n_input)
  train_x, train_y = data[:, :-1], data[:, -1]
  train_x = train_x.reshape((train_x.shape[0], n_seq, n_steps, 1))
  # layers of the model:
  model = Sequential()
  model.add(TimeDistributed(Conv1D(n_filters, n_kernel, activation='relu',
  input_shape=(None,n_steps,1))))
  model.add(TimeDistributed(Conv1D(n_filters, n_kernel, activation='relu')))
  model.add(TimeDistributed(MaxPooling1D()))
  model.add(TimeDistributed(Flatten()))
  model.add(LSTM(n_nodes, activation='relu'))
  model.add(Dense(n_nodes, activation='relu'))
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model

# forecast with a pre-fit model
def model_predict(model, history, configuration):
  n_seq, n_steps, _, _, _, _, _ = configuration
  n_input = n_seq * n_steps
  x_input = array(history[-n_input:]).reshape((1, n_seq, n_steps, 1))
  prov = model.predict(x_input, verbose=0)
  return prov[0]

# walk-forward validation for univariate data
def walk_forward_validation(data, num_test, config):
  global predictions
  predictions = list()
  # split the dataset
  train, test = split_dataset(data, num_test)
  model = model_fit(train, config)
  # feed history
  history = [x for x in train]
  for i in range(len(test)):
        prov = model_predict(model, history, config)
        predictions.append(prov)
        history.append(test[i])
  # estimate the error
  error = measure_rmse(test, predictions)
  print(' > %.3f' % error)
  return error

# repeat to reduce stoch
def repeat_evaluate(data, configuration, num_test, n_repeats=30):
  # fit and evaluate the model n times
  scores = [walk_forward_validation(data, num_test, configuration)
    for _ in range(n_repeats)]
  return scores

def summarize_scores(name, scores):
  # print a summary
  scores_m, score_std = mean(scores), std(scores)
  print('%s: %.3f RMSE (+/- %.3f)' % (name, scores_m, score_std))

series = read_csv('final_dataframe.csv', header=0, index_col=0)
del series['temperature (C)']
val_max=series.max()
series=series/val_max
data = series.values
# to split dataset
```

```
num_test = 100
# define configuration
configuration = [3, 12, 64, 3, 100, 200, 100]
scores = repeat_evaluate(data, configuration, num_test)
summarize_scores('cnn-lstm', scores)

executionTime = (time.time() - startTime)
print('Execution time in seconds: ' + str(executionTime))

series=series*val_max
data=series.values
result=walk_forward_validation(data, num_test, configuration)
prediction_values=list()
for i in range(len(data)-num_test):
  prediction_values.append(data[i])
for i in range(num_test):
  prediction_values.append(predictions[i])
pyplot.plot(prediction_values)
pyplot.plot(data)
pyplot.ylabel('Energy (kWh)')
pyplot.xlabel('Time serie value')
pyplot.show()
```

J LSTM multivariate Code

```
#######LSTM#######
from math import sqrt
from numpy import concatenate
from matplotlib import pyplot
from pandas import read_csv
from pandas import DataFrame
from pandas import concat
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
import time
startTime = time.time()

def series_to_supervised(data, n_in=1, n_out=1, dropnan=True):
  n_vars = 1 if type(data) is list else data.shape[1]
  df = DataFrame(data)
  cols, names = list(), list()
  # input sequence (t-n, ... t-1)
  for i in range(n_in, 0, -1):
        cols.append(df.shift(i))
        names += [('var%d(t-%d)' % (j+1, i)) for j in range(n_vars)]
  # if more than one output is needed
  for i in range(0, n_out):
        cols.append(df.shift(-i))
        if i == 0:
          names += [('var%d(t)' % (j+1)) for j in range(n_vars)]
        else:
          names += [('var%d(t+%d)' % (j+1, i)) for j in range(n_vars)]
  # put it all together
  agg = concat(cols, axis=1)
  agg.columns = names
  if dropnan:
        agg.dropna(inplace=True)
  return agg

dataset = read_csv('canaria_matrix.csv', header=0, index_col=0)
values = dataset.values
values = values.astype('float32')
# normalize
scaler = MinMaxScaler(feature_range=(0, 1))
scaled = scaler.fit_transform(values)
# change to supervised learning
reframed = series_to_supervised(scaled, 1, 1)
# drop columns that will be not predicted
reframed.drop(reframed.columns[[4,5]], axis=1, inplace=True)

# split into train and test sets
values = reframed.values
n_train_days = 265
```

```
train = values[:n_train_days, :]
test = values[n_train_days:, :]
# split into input and outputs
train_X, train_y = train[:, :-1], train[:, -1]
test_X, test_y = test[:, :-1], test[:, -1]
# reshape input to be 3D [samples, timesteps, features]
train_X = train_X.reshape((train_X.shape[0], 1, train_X.shape[1]))
test_X = test_X.reshape((test_X.shape[0], 1, test_X.shape[1]))

# neural network layers
model = Sequential()
model.add(LSTM(50, input_shape=(train_X.shape[1], train_X.shape[2])))
model.add(Dense(1))
model.compile(loss='mae', optimizer='adam')

history = model.fit(train_X, train_y, epochs=50, batch_size=72, validation_data=(test_

prov = model.predict(test_X)
test_X = test_X.reshape((test_X.shape[0], test_X.shape[2]))
# invert scaling for forecast
inv_prov = concatenate((prov, test_X[:, 1:]), axis=1)
inv_prov = scaler.inverse_transform(inv_prov)
inv_prov = inv_prov[:,0]
# invert scaling for actual
test_y = test_y.reshape((len(test_y), 1))
inv_y = concatenate((test_y, test_X[:, 1:]), axis=1)
inv_y = scaler.inverse_transform(inv_y)
inv_y = inv_y[:,0]
# calculate RMSE
rmse = sqrt(mean_squared_error(test_y, prov))
print('Test RMSE: %.3f' % rmse)

inv_prov=list(inv_prov)
inv_y=list(inv_y)
first=list(dataset['energy (MWh)'].values)
pred = first[:n_train_days]
testeado=first[:n_train_days]
for i in range(len(inv_prov)):
  pred.append(inv_prov[i])
  testeado.append(inv_y[i])

executionTime = (time.time() - startTime)
print('Execution time in seconds: ' + str(executionTime))

pyplot.plot(pred)
pyplot.plot(testeado)
pyplot.ylabel('Energy (kWh)')
pyplot.xlabel('Time serie value')
pyplot.show()
```

ETSEIB