# Report 2 : Forecasting and Analysis of the Polish Power System Time Series

Piero Paialunga and Simone Chierichini

October 2020

# Contents

## Abstract

The Load of the Polish Power System Dataset has been analyzed with various methods and a forecast have been made out of this analysis. Two Fourier methods (time independent and time dependent) have been applied to extract information about the periodical behavior of the signal. The wavelet has been used to clean the signal from the noise, and the cleaned signal has been used to apply prediction with ARIMA and SARIMA processes and deconvolution methods. The best forecast gave a RMSE equal to the 9.7% of the maximum value.

# Chapter 1

# Introduction

An appealing kind of dataset that is worth studying in order to obtain a predictive ability about nature are the Time Series. The one studied in this report is about the load of the Polish Power System.

One of the most powerful tool of Data Analysis of a Time Series consists on using the Fourier transform on a periodic data. In fact, the Fourier Theorem states that a periodic function can be decomposed in sines and cosines terms. In this sense it is possible to perform a variable space from the original one (the $x$ space) and analyze the behavior of the function with respect of the $u$ frequency, thus going from $f(x)$ to $F(u)$. In particular, the absolute value of this function computed at a specific frequency $u^*$ furnishes the amplitude that $u^*$ has in the spectrum. By the distinction of the frequencies that could be considered as a form of "noise" and the frequencies of the "proper" signal, and by the accurate treatment of the non-stationary nature of the signal, it is possible to use the information of the frequency spectrum to filter the signal and gain a forecast ability.

Another important tool to clean the signal consists in the use of the Wavelet transform. As the Fourier Analysis, the wavelet transform compute the projection of the original signal on an orthogonal basis of a function called wavelet. By using this wavelet at different scales, it is possible to detect the noise scale as it has frequencies that can be assumed to be above the typical band of frequency of the signal.

A predictive method that is more powerful of the Fourier Analysis consists in the use of ARIMA and SARIMA processes. By the use of this processes it is possible to construct an approximated "auto-regressive" and "moving average" model of the original signal. In particular, the main difference between this method and the other ones that has been used is the ability to use a different numbers of values to look at both in the auto-regressive and the moving-average processes to obtain a prediction. Moreover, in the SARIMA processes, the non stationary nature of the signal could be treated with an extra-attention as the method considers the seasonal elements of the data.

The last method that has been used to forecast and analyze the signal is the deconvolution. This approach considers the signal as the convolution of one single shape known as kernel with a series of pulses. A refinement of this method has been applied considering three different kernels.

# Chapter 2

# Data and Method Description

## 2.1 Data overview

The analyzed data is the load of Polish Power System time series. The dataset is extremely simple as it consists in a time report of all the loads in the Polish Power System. The columns of the dataset are:

- **Day**, Calendar Day in the following format ('YYYY/MM/DD')
- **Hour**, Hour of the end of the detection
- **Minute**, Minute of the end of the detection
- **Load**, Load (MW) during that time interval

Measurements are taken each 15 minutes and the Load column reports, in the $(n)$-th row, the load between the $(n-1)$th and the $(n)$th row (i.e. in 15 minutes of time). The site that has been used to extract the dataset uploads real time information about the Polish Load. Due to computational limits, our specific dataset is limited from 2008 (2008-01-01) to 2016 (2016-12-31).

## 2.2 Data preprocessing

Even if the initial format of the dataset is imediate to read and to understand, it is convenient to convert the temporal scale in a way that the temporal continuity is visible and appreciable. To do so, a 5-th column has been added to the dataset, consisting in a temporal line from 900 seconds (15 minutes) and increasing by the same factor (900 seconds) for each row: (900,1800,2700,...). Using this temporal scale, an effective visualization of the load can be obtained:

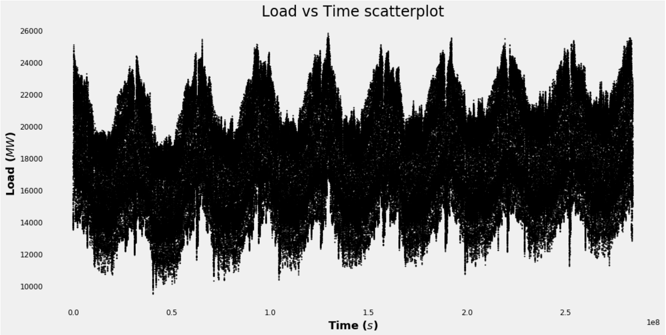


Figure 2.1: **Load($MW$)vs Time($s$) plot**.

The analysis we want to perform about the dataset is based on the Fourier Transform. As it has already been said, the goal of this analysis is to detect the frequencies that characterize the phenomenon of interest. In this sense, the mean value and the global temporal trend of the signal are not interesting and tend to disturb the target analysis. As it can be appreciated from Figure 2.1 the mean value of the signal is not 0 and a global linear ascending trend can be easily identify. The first step is thus "detrending" the original signal, and the final result is expressed in Figure 2.2:

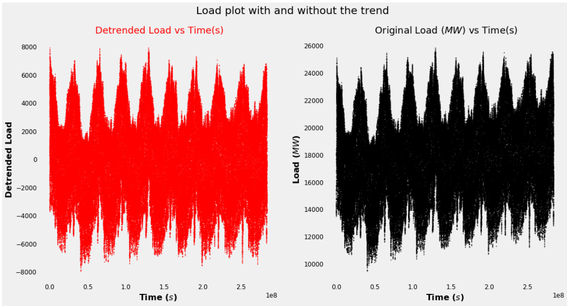


Figure 2.2: **De-trended Load vs Time($s$) plot** to the left and Load ($MW$) vs Time ($s$ plot to the right.

The mean value of the red signal is extremely low ($\approx 10^{-11}$). To make it even lower, the mean value is subtracted from the red signal, thus obtaining a signal with mean value$= 0$ (numerically the mean value is the lowest possible: $\approx 10^{-13}$). From now on, the "detrended" mean=0 Load will be simply intended as load. As it has already been said in chapter 1, the Fourier Analysis is made for periodical signal. The first important thing to notice is if the first point ($i = 0$) and the last point ($i = N$) of the dataset has the same Load quantity. As there is in fact a notable difference between $Load_0$ and $Load_N$ ($|Load_0 - Load_N| = 2267.67$ that is almost the 30% of the maximum load), a window function needs to applied to smooth the signal and make the endpoint of the signal meet. The first standard windowing function that has been applied to do so is the Hanning function [6] and the result of the product between the signal and the hanning function is shown in Figure 2.3

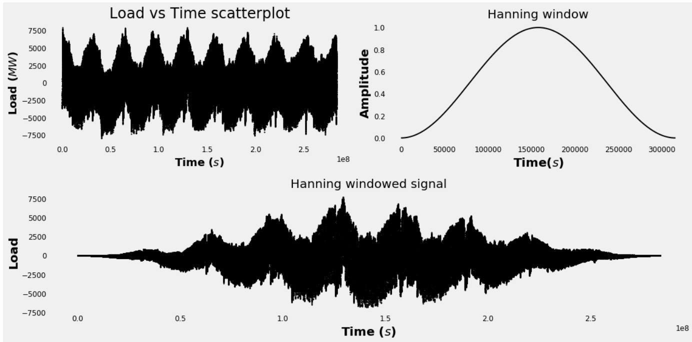


Figure 2.3: **Detended Signal Load vs Time scatteplot (up-left), Hanning window (up-right) and their product windowed signal**. As it is possible to see, the windowed signal has the same value on its border.

## 2.3 Fourier Transform

As it has already been said in the introduction, the signals that can be analyzed using Fourier Transform are by definition stationary. This condition appear to be a forced one when the signal is extracted by some real world data like the one that it has been considered. For example the behavior of the Load of a civilized country during the Christmas Holidays is not equal to the behavior during the rest of the year. For this reason, the blind application of the (Fast) Fourier Transform may seems a weak approach. Nonetheless some basic time period could still be verified from the Fourier transform of the Load. In fact by just looking at the plot Amplitude vs Period ($h$) it is possible to notice some natural order periods like:

- 12h (the lenght of a day divided by 2)
- 24h (the lenght of a day)
- $\approx$ 33h (1 day + 8h )
- 168h (7 days, a week)

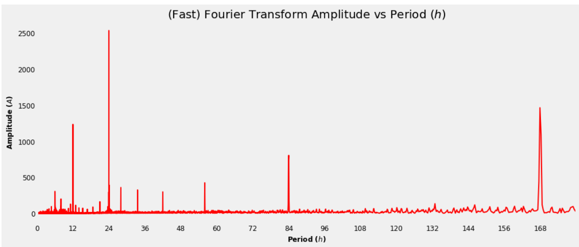


Figure 2.4: **Amplitude vs Period ($h$) plot.** As it is possible to notice some relevant peaks appear in some specific time ticks (12h,24h,33h,168h).

This specific periods seem to be high correlated with the working hours and the natural duration of a day. In fact, it is possible to assume that the 12h periodicity could be related to the day and the night hours of a day. 24h is the duration of a day, so it is still a natural period. 33h is the sum of 24h (the lenght of the day) + 8h (the mean working hours per day) and it could be related to some working periodicity. Even the week and the middle week periods are stressed by the 84h and 168h peaks. Thus, even if the signal is not stationary and some extra work needs to be done in order to interpret the signal correctly and thus to be able to predict the following year, **it is still important to analyze this periodicity and understand if it is possible to reconstruct the signal starting with few frequencies values**. As the numpy algorithm that has been used is a robust one, transforming the original signal using Fourier transform and then inverse transforming the Fourier transform, the original signal is accurately reproduced. The mean absolute difference between the reconstructed signal and the original one is in fact almost 0 ($\approx 10^{-13}$) and the signal are almost unrecognizable from each other as it is possible to see from Figure 2.5:

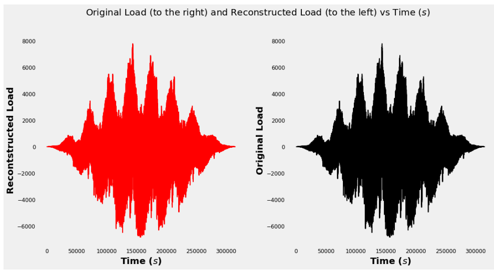


Figure 2.5: **Reconstructed (to the left) and original signal (to the right) comparison.** As it is possible to see, the signal are unrecognizable from each other, as the numpy FFT algorithm is robust.

After that the robustness of the algorithm has been tested, it is interesting to analyze which frequencies are really informative in terms of the reconstruction of the original signal. In particular, a threshold has been applied to the frequency spectrum, setting to 0 the values that are greater than that threshold. After this operation the inverse fourier transform has been applied, thus obtaining a reconstructed signal. The root mean squared error (RMSE) has been computed between the original signal and the reconstructed one.

### 2.3.1 Uniform varying threshold

Varying the threshold ($k$) in an uniform manner ($k \in \{0, 2475\}$) the RMSE becomes larger as one could expect. To deeply understand the meaning of this RMSE and how the information is filtered out by the threshold value, some $k$ values filtered plot has been shown in Figure 2.6

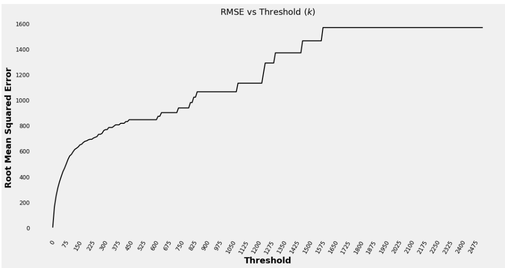


Figure 2.6: **RMSE vs $k$ plot.** As the threshold ($Th_k = k$) increases the RMSE increases too, thus highlighting the loss of information that it is verified during the reconstruction. The highest growth rate is in the first zone of the graph, where $k \in \{0, 300\}$

As Figure 2.7 suggests, as the $k$ value increases, only the frequencies with highest amplitude are not filtered out. In fact, when $k$ becomes bigger than a certain value, only the highest peak, that is the one related to the 24h period, "survives".

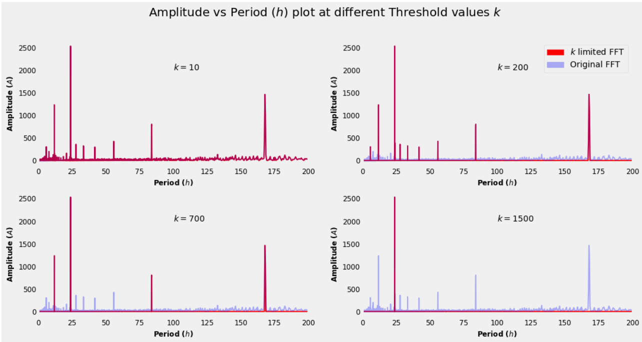


Figure 2.7: **FFT of the original signal and k-filtered FFT vs Period ($h$) plot.** As the threshold ($Th_k = k$) increases only the frequencies with the highest amplitude are not filtered out.

### 2.3.2 Maximum related threshold

The analysis suggests that,to understand the relevance of the sinusoidal components, it could be more convenient to vary the threshold $k$ scale with respect to the maximum amplitude value. As it is possible to see from Figure 2.8, if the frequencies with amplitude under the 30% of the maximum are set to 0, a big portion of the noise is removed, and the waving behavior of the signal is still visible and cleaned. As the threshold becomes higher, only the 24h period survives. Its frequency, as it is possible to see from Figure 2.4, is in fact so high to be indistinguishable from a full colored square, as the last subplot in Figure 2.8 highlights. A clearer version of Figure 2.6 has been reproduced, computing the RMSE at the varying of the $k$ with respect to the maximum amplitude value (Figure 2.9). Even if, in general the RMSE furnishes an efficient comparison method be-

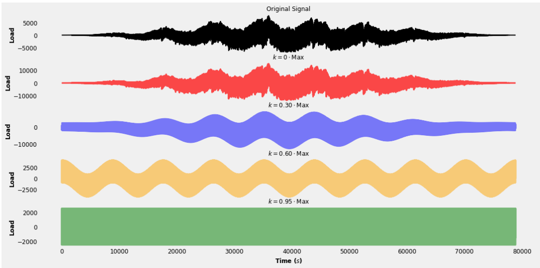


Figure 2.8: **Reconstructed Load vs Time at various $k$ threshold value.** If $k = 0.3{\cdot}\text{Max}$, the original signal can be reconstructed with a discrete level of accuracy.

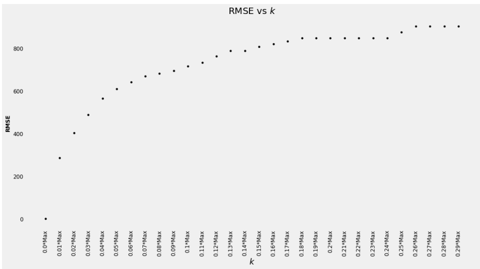


Figure 2.9: **RMSE vs $k$ plot**. This plot is analog to the one reproduced in Figure 2.6, but it is simpler to read as it is related to the maximum amplitude value.

tween a real signal and a reconstructed one (low RMSE implies good correspondence between the two), it is important to understand if the RMSE value is dominated by the background noise of the original signal. In fact, while reconstructing the signal with the threshold limited fourier transform, the noisy behavior of the original Load is set to zero and a more clean signal is reconstructed (example in Figure 2.10). This process summarize the purpose of this part of the report, as it cleans the original data and set to zero the not relevant information, but increases the RMSE.

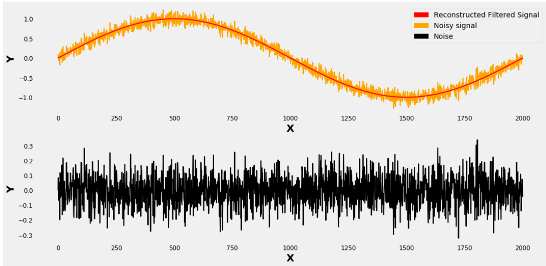


Figure 2.10: **Example of the noise phenomenon**. Filtering the fourier spectrum with a threshold, the noisy sinus wave looses its background noise. As it is possible to see, the noise is not related to the original signal.

To explore this phenomenon, it is important to understand how much the error is related or not to the original signal. The plot of the correlation value with respect to $k$ predictably highlights that the correlation increases with the $k$ threshold. The $k$ value can thus be chosen with respect to this correlation coefficient. It is important to highlight the trade off that needs to be obtained here:

- An high threshold value assures that a rigid selection of the frequencies has been applied and the noise is deleted
- A low correlation coefficient assures that the residual signal is not correlated with the original signal and it can be properly assume as noise.

As it is possible to see from Figure 2.11, as the correlation decreases the threshold decreases too and vice versa. It is thus important, in order to delete only the signal, to keep the threshold as high as possible while the correlation doesn't increase over a certain value. As it is possible to appreciate from Figure 2.11, the correlation between the error and the signal is not 0 even when the threshold is set to 0, thus highlighting that even if the error that is generated by the Fourier Transform is extremely low it is still slightly correlated with the signal ($C = 4.3\%$ correlation). This

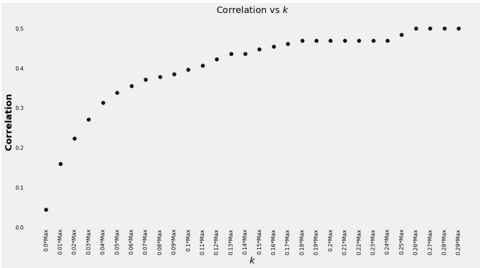


Figure 2.11: **Correlation vs $k$ scatter plot**. As it is possible to see from the plot, as the $k$ threshold value increases, the error becomes more correlated with the original signal.

consideration, together with the correlation instantly increasing in the first 3 thresholds ($C_{k=0.05*Max} = 16\%$,$C_{k=0.02*Max} = 23\%$,$C_{k=0.03*Max} = 27\%$), furnishes a relevant warning to further decrease the $k$ lower bound. As it is possible to see from Figure 2.12 **this detailed observation permits to choose a less correlated signal reconstruction by setting the optimal threshold as the highest one with correspondent correlation lower than** $10\%$ ($k_{opt} = 0.004 \cdot Max$).
The optimal signal reconstruction is thus obtained by inverse transforming the filtered power spectrum and the results are shown in Figure 2.13.

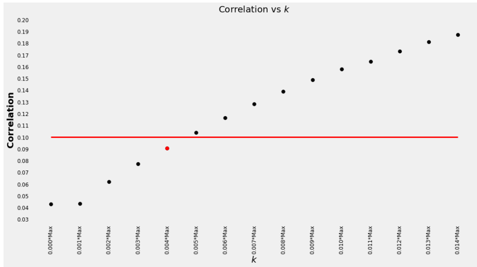


Figure 2.12: **Correlation vs $k$ scatter plot**. As it is possible to see from the plot, as the $k$ threshold value increases, the error becomes more correlated with the original signal. The optimal threshold

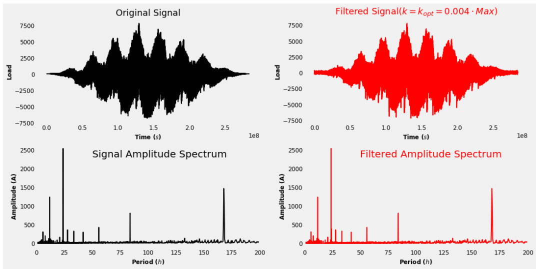


Figure 2.13: **Original (Up-left) and Filtered (Up-right) Signal with correspondent amplitude spectra (Down-left and Down-right)**. As it is possible to see, the signals appear similar but does have some differences in the highest frequencies.

A double check of the analysis has been done to verify the distribution plot of the difference between the reconstructed signal and the original one. In fact in order to be considered white gaussian noise, it is expected to be fitted with low $\chi^2$ value to a gaussian [5]. As it is possible to see from Figure 2.14 the error appears to have a gaussian shape. It is thus interesting

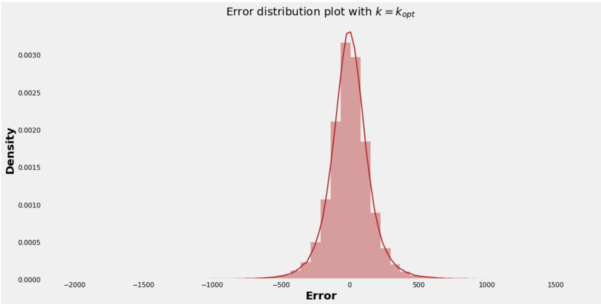


Figure 2.14: **Error Distribution plot with $k = k_{opt}$.** As it is possible to see, the distribution has a gaussian shape in its core.

to perform the $\chi^2$ test on a gaussian distribution, as it has been shown in Figure 2.15. Unsurprisingly, the $\chi^2$ value is low $\chi^2 = 6.27 \times 10^{-2}$ while the p value [7] is extremely high $p \approx 1.0$, thus highlighting the excellent correlation between the gaussian fit and the original data. Even if this

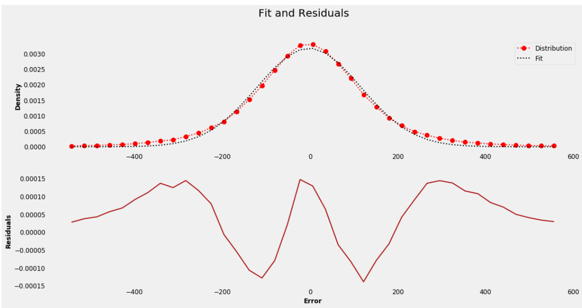


Figure 2.15: **Error Distribution plot with $k = k_{opt}$ and gaussian fit (Up), Residuals plot (Down).** As it is possible to see, the gaussian fit with high accuracy the distribution plot. The residuals $(R)$ are at a significant lower scale with respect to the distribution values $(D)$: $\frac{D_{max}}{R_{max}} = 22.360$

method shows promising statistical properties, **the important test that the reconstructed signal has to pass is the prediction of the new data**. As it is important to highlight as a background clarification, **the prediction that can be done is about the oscillatory behavior of the core of the signal**. In fact, the linear trend (the growth of the signal) and the mean value are both subtracted from the original signal in order to process it correctly with the Fourier analysis. Moreover, two important hypothesis that has been made are the following:

- **The nature of the signal is periodical**: all the events can be accurately modeled as sines and cosines terms.
- **The signal is stationary**: the frequencies of the fourier spectrum are time invariant.

While the first assumption is a crucial one when the Fourier transform is applied, the second one will be relaxed with the application of the **spectrogram method**. In order to perform the prediction test, the dataset has been split in two sets. Borrowing in a not rigorous manner the Machine Learning notation they have been defined as following:

- **The training set**, where the algorithm that has been described in the previous page has been applied and the prediction has been performed.
- **The test set**, that is the set of data that has been used for testing the prediction.

In order to apply the algorithm in the most efficient way as possible, the training set has been obtained by extracting the first 8 years of the dataset, while the remaining year (2016) has been used as test set. The previous described algorithm has been applied in the training set, but as it is necessary to predict the real values no smoothing has been applied to the original signal. In order to keep both RMSE and Correlation values under control ($RMSE \leq 300$, $C \leq 10\%$), the optimal threshold as been reduced : $k_{opt} = 0.004 \cdot Max$. The plot of both RMSE and Correlation values are reported on Figure 2.16 while the spectrum with the $k_{opt}$ threshold has been reported in Figure 2.17.

The spectrum in Figure 2.17 furnishes a

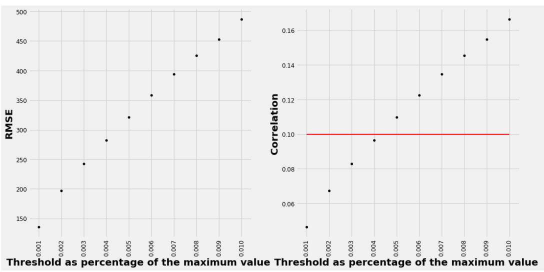


Figure 2.16: **RMSE and Correlation vs $k$ plot)**. In order to keep $RMSE \leq 300$ and $C \leq 10\%$, the $k_{opt}$ value has been chosen to be $k_{opt} = 0.004 \cdot Max$.

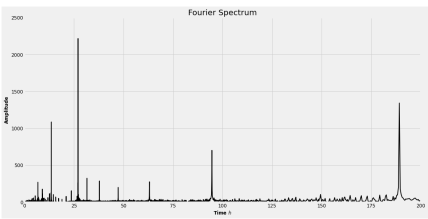


Figure 2.17: **RMSE and Correlation vs $k$ plot)**. In order to keep $RMSE \leq 300$ and $C \leq 10\%$, the $k_{opt}$ value has been chosen to be $k_{opt} = 0.004 \cdot Max$.

set of sines and cosines that can be defined by their frequency $\omega$ and their complex correspondent value ($F(\omega) = F_R(\omega) + jF_I(\omega)$). Given a specific frequency $\omega$ and a specific time $t$ the correspondent waving function is given by:

$$x(t, \omega) = \frac{2}{N}(F_R(\omega)\cos(2\pi\omega t) + F_I(\omega)\sin(2\pi\omega t))$$

In general, for all the frequencies that has not been filtered out ($\omega \in \Omega$) we have:

$$x(t) = \sum_{\omega \in \Omega} \frac{2}{N}(F_R(\omega)\cos(2\pi\omega t) + F_I(\omega)\sin(2\pi\omega t))$$

By applying this rule with $t \in T_{train}$ with $T_{train}$ that is defined by all the time steps ($s$) of the training data, the reconstructed signal is simply the inverse (filtered) fourier transform of the original one. Nonetheless, when $t \notin T_{train}$ but $t \in T_{test}$ the signal is extended over its inverse fourier definition by using the above rule. In this sense, as the algorithm "didn't see" the test set, a prediction has been made. As the reconstruction has been done on the training set frequencies, the difference between the original signal and the reconstructed one is at its minimum values in the training set and it increases on the test set. Indeed, it is possible to see that the increase of the difference between the predicted signal and the real one doubles in the test set.

In particular the RMSE in the training set is:

$$RMSE_{Train} = 278.32$$

While the RMSE in the test set is:

$$RMSE_{Test} = 2624.89$$

This result comes with little surprise. In fact no temporal information has been given

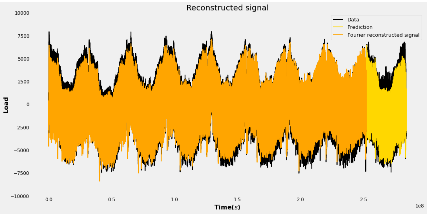


Figure 2.18: **Original Signal, Reconstructed Signal and Predicted Signal vs Time($s$)**

about the frequencies and the signal is considered to be stationary. A close look of the original signal and the reconstructed one in the test set highlights that the errors of the prediction become significantly higher with respect to the time.

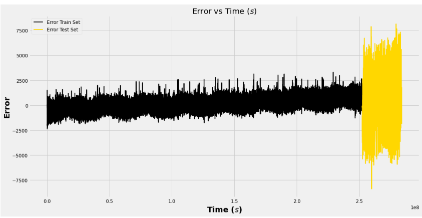


Figure 2.19: **Error vs Time**. The error becomes consistently higher when the test set is explored. In both training set and test set the signal appears to be slightly correlated to the error.

Moreover **the error increases with respect to the time, thus highlighting the weaknesses of the stationarity assumption**. As it has already been told, the RMSE could be a inappropriate metric. A deeper analysis has been made to see whether or not high correlation could be found between the original signal and the error. Unfortunately the signal and the error are highly anticorrelated ($C_{data,error} \approx -0.6$) and the $P$-value test highly rejects the gaussian hypothesis ($P \approx 0$), thus confirming the failure of the method. The

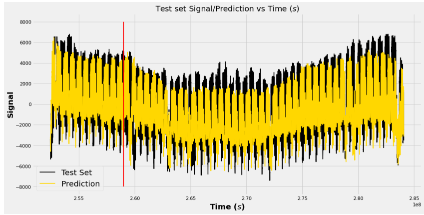


Figure 2.20: **Signal/Prediction vs Time** ). It is possible to see that the prediction becomes worse at high time scale, while it seems to be acceptable for the first entries.

RMSE has been computed reducing the time range and it can be seen that the first prediction have the lowest error values. In particular the prediction for the first 30 days is both the most precise one in terms of RMSE and the less error-related. As the

| | $T_f in$(s) | RMSE | $\Delta T$(s) | $\lvert C_{data,error} \rvert$ | $\Delta Days$ |
|---|---|---|---|---|---|
| 0 | 254188800 | 2569.99 | 1727100 | 0.64 | 20 |
| 1 | 255052800 | 2439.43 | 2591100 | 0.61 | 30 |
| 2 | 257644800 | 2458.33 | 5183100 | 0.62 | 60 |
| 3 | 260236800 | 2780.33 | 7775100 | 0.65 | 90 |
| 4 | 262828800 | 2686.71 | 10367100 | 0.64 | 120 |

Figure 2.21: **Summary of the RMSE at different time ranges)**. As it is possible to see, the prediction gets better for lower values of time step. In particular the best RMSE and correlation (the lowest) are verified for a 30 days time step.

(best) RMSE is the 33% of the maximum value of the signal, the method has to be considered as failed.

### 2.3.3 Time dependent Fourier Transform

The algorithm that has been presented in Chapter 2.3.2 had the important defect of being completely time independent. In fact the signal has been interpreted as a stationary one, thus obtaining a weak reconstruction and an even weaker prediction. The following approach [4] is used to consider the non stationary nature of the dataset, and it is based on the following division of the original data:

- **The training set**, that contains the signal from 2008 to 2015
- **The validation set**, that contains the signal data of the year 2015
- **The test set** that contains the signal data of the year 2016

The training set has been divided in 5 annual periods. **For each one of those, the fourier transform has been applied. The mean of all these transform has been applied, thus obtaining a mean fourier transform of the first 5 years of the dataset**.
The validation set has been used to apply the algorithm that has been described in chapter 2.3.2 to the mean fourier transform that has been obtained from the training set. **The big difference between this approach and the one described in chapter 2.3.2 is that the RMSE and the correlation are computed with respect to a portion of the dataset that the algorithm "does not know": the algorithm is not been trained on this portion of the dataset**. To consider this difference, the RMSE threshold and the correlation one are less strict than the one consider in chapter 2.3.2 ($RMSE_{max} < 2000$,$|C_{min}| < 0.82$).
The threshold value with the lowest $C_{min}$ and within the $RMSE_{max}$ value has been considered as the **optimal threshold**. **The mean fourier transform that has been computed in the training set has thus been filtered with the optimal threshold, and the result has been compared with the test set signal**. These has been summed together and divided by 8, thus obtaining a mean fourier transform that has been adopted to predict the values of the 9th year.

| | RMSE | Days | $C_{data,error}$ |
|---|---|---|---|
| **0** | 1693.717665 | 10 | -0.499780 |
| **1** | 1653.486625 | 20 | -0.584994 |
| **2** | 1625.299205 | 30 | -0.551923 |
| **3** | 1750.071345 | 40 | -0.572060 |
| **4** | 1705.124815 | 50 | -0.576772 |
| **5** | 1742.991733 | 60 | -0.582522 |
| **6** | 1739.389256 | 70 | -0.587244 |
| **7** | 1757.974530 | 80 | -0.591003 |
| **8** | 1945.728359 | 90 | -0.620888 |

Figure 2.22: **Summary of the RMSE after a certain number of days of observation**. Even if the RMSE is in general still remarkably high the values are lower than the ones that have been obtained with the previous method. Even the correlation values are lower than the ones of the previous method.

This method remarkably outclass the previous one in terms of RMSEs, obtaining lower RMSEs even for larger period of time. The method is better in terms of correlation between the original signal and the error too. In fact data are almost 10% less correlated with the error with respect to the previous method.

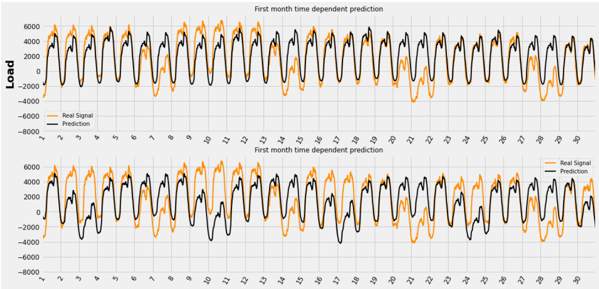


Figure 2.23: **First month prediction using Time dependent Fourier Transform method)**.

**The best RMSE assumes a value that is the 21% of the maximum value thus obtaining a more accurate method with respect of the (filtered) global fourier transform one**.

## 2.4 Wavelet

A wide part of the algorithm in section 2.3.2 and 2.3.1 is based on cleaning the original signal from the (presumed) noise. **In general, an important step in order to have an accurate prediction of a time-series is based on cleaning the original data**. So far, this cleaning process has been done following the step indicated above:

- Analyzing the fourier spectrum
- Applying a threshold to it
- Selecting the upper bound of the threshold based on the RMSE
- Selecting the optimal threshold as the highest one with the correlation value as close as possible to 10%

This methodology has not been proved helpful even if it is based on the reasonable assumption that the residual signal has to be not correlated with the signal to be considered as noise. A different approach to clean the signal is based on using the so called **wavelet transform**. **The wavelet transform** is in some way similar to the Fourier one as it **computes the product between the original signal and a basis function** (**$\psi$**). Nonetheless this basis function is stretched and contracted in relation with a parameter called **scale** ($s$). Thus, at a fixed scale $s$, **the basis function will "slide" on the original signal time steps outputting the projection between the signal $f$ at that specific time and the wavelet computed at that specific scale and that specific time**. The sum of all this time contribute will give the wavelet coefficient at a specific time ($u$) and at a specific scale ($s$):

$$W_f(s,u) = \int_{-\infty}^{+\infty} f(t)\frac{1}{\sqrt{s}}\psi^*(\frac{t-u}{s})dt$$

The power of this method relies on the ability of discern and decompose the original signal at different scale. Indeed, when the scale decreases, higher frequencies of the original signal are detected and vice versa. By using low value of scale, the wavelet method is able to discern the highest frequencies of the signal, that are the one that can be considered to belong to the white noise frequency range. In particular, the method that has been used to implement this idea is the **discrete wavelet transform** [2]. This method is based on the decomposition of the signal in **detail** and **approximation coefficients** on various level. The first level is obtained by using the wavelet with the lowest possible scale: i.e. the highest frequency ($\nu_{s_{min}}$) with respect to the highest frequency of the signal ($\nu_{max}$) according to the Nyquist Shannon theorem ($\frac{\nu_{max}}{2} = \nu_{s_{min}}$) [1]. In particular, **the detail coefficients of the first level are represented by $W_f(s_{min}, u)$ and the approximation coefficients are represented by the residual between the original signal and the detail coefficients**. Both the detail and the approximation coefficients, according to the Nyquist Shannon theorem can be downsampled by a factor $2^{n_{level}}$ (2 for the first level, 4 for the second level, 8 for the third level). **As it can be assumed that the denoised signal will have specific band limited frequency range, the highest frequencies of the studied signal can be considered as the white noise frequencies** [3]. That means that it is possible to detect the noise signal in the lowest level of the discrete wavelet transform detail coefficients.

The first level presents a not negligible excess of kurtosis, thus discouraging from looking at other levels, as too much relevant information about the signal would be lost. Moreover, in order to select the part of the detail coefficient that can't be considered as "noise", a certain portion of the detail coefficient has been set to 0. This portion has been chosen with respect of a threshold value $\theta_k$ that is a function of $k$, that is a real number between 0 and 13.25 ($k \in [0, 13.25]$) and the fitted sigma value from the gaussian fit:

$$\theta_k = k\sigma$$

This value has to be intended as a threshold in the following sense: **all the values of the details coefficient that are found between $-\theta_k$ and $+\theta_k$ are set to 0**. The $k$ upper bound is due to the fact that if the threshold is chosen to be that high, the entire first detail coefficient is set to 0. While $k$, (and $\theta_k$) increases, so does the RMSE. Indeed if $\theta_0 = 0$ the approximation and detail coefficient are naively summed, as no value

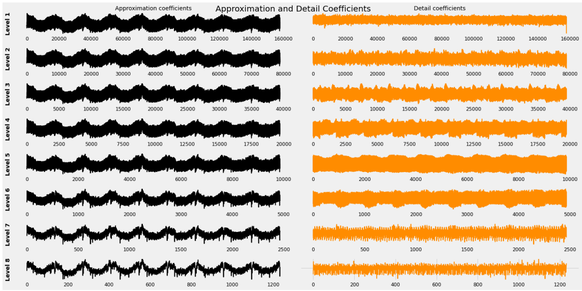


Figure 2.24: **Approximation and Detail Coefficients)**. As it is possible to see, the periodical behavior of the signal does not present itself in the first detail coefficients, while it starts appear when the $n_{level}$ increases. At the same time the approximation coefficients are really similar to the original signal for the first level, and they loose accuracy while the level increases.

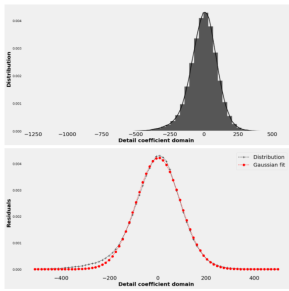

Figure 2.25: **Detail Coefficient distribution)**. As it is possible to see, the distribution is almost gaussian, but presents an excess of kurtosis on its tail.

of the detail coefficient is set to 0. Nevertheless, if the RMSE increments are due to the noise they are harmless as they don't imply a loss of information about the original signal. **In order to detect whether or not the filtering is actually cutting only the noise out, the correlation coefficient between the original signal and the difference between the latter and the reconstructed one (intended as the sum of the approximation coefficient and the filtered detail coefficient both of the first level) has been computed. The best threshold has been chosen to be the one with the lowest correlation**. In particular it has been proven to be 1:

$$\theta_{opt} = \sigma$$

. This optimal threshold permitted to have the following error-signal correlation

$$C = 0.2\%$$

The effect of the threshold on the original detail coefficient domain has been shown in Figure 2.27. This optimal threshold has been used to construct a reconstructed signal, that is as less as possible influenced by the noise. As it is possible to see from Figure 2.28, it is really difficult to spot the differences between the original signal and the reconstructed one in general terms. On the other hand the smoothing effect can be appreciated at lower scales, where the original signal and the reconstructed one are still similar, but the reconstructed signal follows a smoother line as it is not 'disturbed' by the noise.

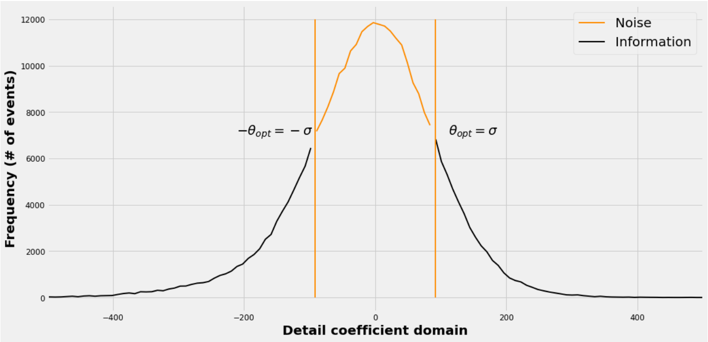


Figure 2.26: **Detail Coefficient distribution filter)**. In orange it is possible to see the range of points that has been filtered out from the original detail coefficient signal. In black, it is possible to see the remaining part, that has been summed to the approximation coefficient signal to obtain the reconstructed filtered signal.

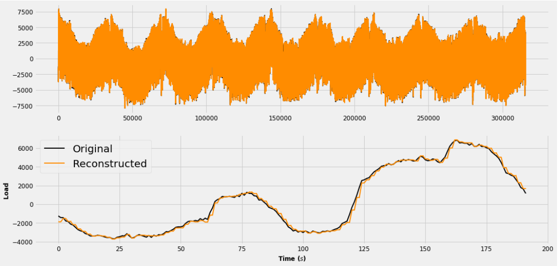


Figure 2.27: **Reconstructed Signal (Orange) and Original Signal (Black) at different time scales**. While the signals appear to be indistinguishable in the larger time scales, the effect of the de-noising wavelet process appears to be visible at lower scales.

## 2.5 Arima and Sarima Processes

In this report we wanted to test the ability of a class of models widely used in forecasting, the Autoregressive–moving-average (ARMA) models. The basic idea is to use these models to describe the dataset and extrapolate a forecast.

The main goal in time series analysis is identifying an appropriate stochastic process that has trajectories that adapt to the data, in order to then be able to formulate forecasts. An important class of stochastic processes that allows us to uniquely identify the process and obtain a consistent estimate is represented by stationary processes. In particular, if what we observe is interpreted as a finite realization of a stochastic process that enjoys particular properties, then it is possible to find a single model suitable to represent the temporal evolution of the phenomenon under study. Intuitively, a stochastic process is stationary if its probabilistic structure (average value, variance, etc.) is invariant over time. Generally a stochastic process $(X_t)_{t\in\mathbb{Z}}$ is said to be *stationary* or *weakly stationary* if it satisfies the following conditions:

- 1. E$[|X_t|^2] < \infty$
- 2. E$[X_t]$= m $\forall t \in \mathbb{Z}$
- 3.$\gamma_X(r, s) = \gamma_X(r + t, s + t) \forall r, s, t \in \mathbb{Z}$

Considering a white noise with zero mean and variance $\sigma^2$ identified as WN(0, $\sigma^2$) and introducing the delay operator B defined as

$$BX_t = X_{t-1} \tag{2.1}$$

we can begin to define some of the fundamental characteristics of these processes. An **Autoregressive process** of order $p$, identified as AR (p) can be written as:

$$x_t = \phi_1 x_{t-1} + \phi_2 x_{t-2} + ... + \phi_p x_{t-p} + w_t \tag{2.2}$$

formally

$$W_t = \phi_0 X_t - \sum_{i=1}^{p} \phi_i X_{t-i} = \phi(B) X_t \tag{2.3}$$

In particular, $\phi$(B) is a polynomial of degree p, given by

$$\phi(z) = \phi_0 - \phi_1 z - ... - \phi_p z^p$$

Where $x_t$ is stationary and $\phi_1, \phi_2, ..., \phi_p$ are constant ($\phi_p \neq 0$). It is generally assumed that $w_t$ is a Gaussian white noise with zero mean and variance $\sigma^2_{w_t}$.

In a very similar way it is possible to define a **Moving Avarage** process of order q, with q $\in \mathbb{N}$, briefly MA(q), if it is of the form:

$$X_t = \theta_0 + \theta_1 W_t + \theta_2 W_{t-1} + ... + \theta_q W_{t-q} =$$

$$= \sum_{i=1}^{q} \theta_i W_{t-i} = \theta(B) W_t$$

where $\theta$(B) is the polynomial of degree q, given by

$$\theta(z) = \theta_0 + \theta_1 z + ... + \theta q z^q$$

A stationary process $(X_t)_{t\in\mathbb{Z}}$ is called *AutoRegressive Moving Average*, more briefly ARMA(p, q), if there are coefficients $\phi_1, ..., \phi_p, \theta_1, ..., \theta_q$ such that

$$x_t - \phi_1 x_{t-1} - ... - \phi_p x_{t-p} = w_t + \theta_1 w_{t-1} + ... + \theta_p w_{t-q} \tag{2.4}$$

Using the polynomials AR and MA defined above and the delay operator B, we can write the equation in compact form:

$$\phi(B) X_t = \theta(B) W_t$$

These models are widely used to study and model time series in particular they are very much related to stationary processes due to their nature. Furthermore, they are very simple models which, however, manage to capture complex data behaviors. However, the specific dataset we have analyzed is very extensive and we have decided to test the power of these models on a **monthly average** of the original dataset, for two reasons: first of all this would have considerably reduced the computational complexity and would have allowed us to compare different configurations, but also because in order to be able to describe all the facets of the original dataset, the model would have required an enormous amount of parameters, and this would have been in contradiction with the nature of these models.

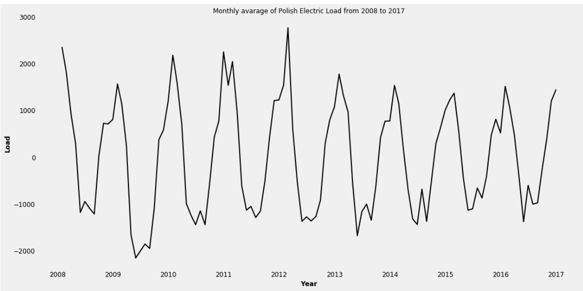


Figure 2.28: Monthly averege of original dataset

### Arima modeling

The first step in analyzing the time series is to check its stationarity, and it was done using Augmented Dickey–Fuller test which tests that the time series can be represented by a unit root(has some time-dependent structure). The alternate hypothesis (rejecting the null hypothesis) is that the time series is stationary. The values of the parameters relating to this test are shown in the following table under the "monthly average" column.

| Differencing | ADF Statistic | P value |
|---|---|---|
| monthly average | -2.64 | 0.085 |
| 1st order | -2.27 | 0.18 |
| 2nd order | -10.99 | $6.80\times10^{-20}$ |

This shows us that according to the criteria of the hypothesis test used, it is necessary to reject the null hypothesis, i.e. that the series is stationary. Table 2.5 also shows the values of the test parameters for the first and second differencing order; In order to make the series stationary, the series has been differencied up to 2 times.

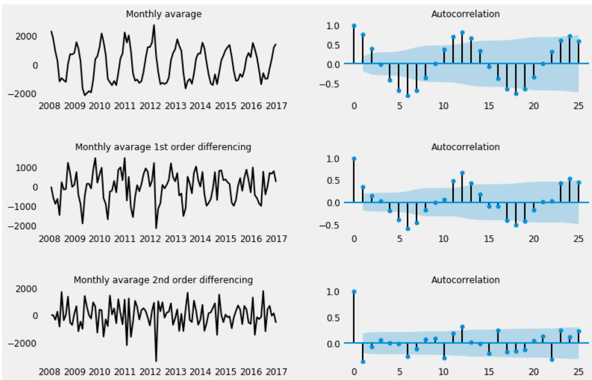


Figure 2.29: Lineplot and autocorrelation plot for the time series, for the 1st. and 2nd order differencing of the Monthly average of Polish Electric Load from 2008 to 2017.

The test is actually successful only for the second order differencing, moreover Figure 2.29 shows how differentiation manages to regularize the series by making it stationary, in particular, the autocorrelation plot for the 2nd differencing is very similar to the autocorrelation of a white noise; These facts would suggest that the differentiation did the trick! However, a closer look at the autocorrelation plot for the 2nd differencing the lag goes into the far negative zone fairly quick, which indicates, the series might have been over differenced. Considering what has been said above, we decided to build two ARIMA models and compare them, one with a differentiation parameter d equal to 0 and the other equal to 2. The parameters relating to the auto-regression p and to the moving average q were selected by comparing the Akaike Information Criterion (AIC) of different models and have been fitted to the data The main information relating to the two models has been reported in the following table:

| Coeff | d=0 | d=2 |
|---|---|---|
| ar.L1 | 0.0819 | 0.5279 |
| ar.L2 | 0.1028 | -0.2791 |
| ar.L3 | . -0.2692 | -0.0252 |
| ar.L4 | 0.3284 | 0.3296 |
| ar.L5 | -0.3148 | -0.5174 |
| ar.L6 | -0.7677 | -0.3888 |
| ma.L1 | 0.3635 | -1.9547 |
| ma.L2 | 0.1579 | 1.3792 |
| ma.L3 | 0.4566 | -0.5751 |
| ma.L4 | -0.2928 | -0.3030 |
| ma.L5 | 0.4209 | 1.1751 |
| ma.L6 | 0.8783 | -0.7160 |
| ma.L7 | 0.2458 | Na |
| ma.L8 | 0.3423 | Na |
| AIC | 1385.880 | 1382.601 |
| BIC | 1423.707 | 1415.099 |

In Table 3 are shown values of autoregressive and moving avarage parameters of the models; in particular the comparison of the AIC value for the two models suggested that the best configuration for the integrated one was ARIMA(p=6, d=2, q=6) and ARMA(p=6,q=8). Furthermore, it is interesting to note that the two models differ very little in terms of AIC and BIC score. The models were built using a statsmodel function called SARIMAX. One of the built-in methods of this function allows to get a diagnostic plot of the model which is very useful to get an idea of the agreement between models and data.

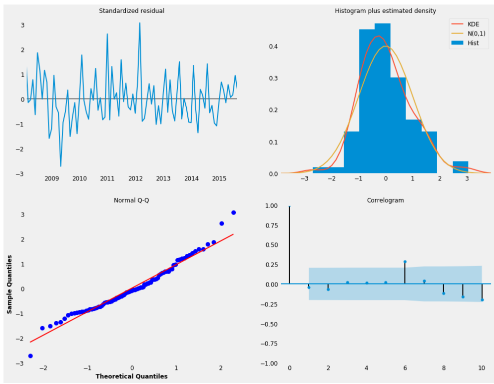


Figure 2.30: Diagnostic plot of the ARIMA(6,2,6) model. The plots show a good match between the model and the data as the residuals are very similar to Gaussian white noise

Figure 2.30 shows the agreement that exists between the ARIMA(6,2,6) model and the data and suggests that the chosen model could be an acceptable representation of the process represented by the data (Diagnostic for the ARMA(6,8) is very similar). Once the information about the two models is gathered, it's time to see how they perform in the forecast; For both models, a training set equal to about 80% of the volume of the series was used to calculate the parameters and was asked to predict the last recorded cycle (still monthly average).

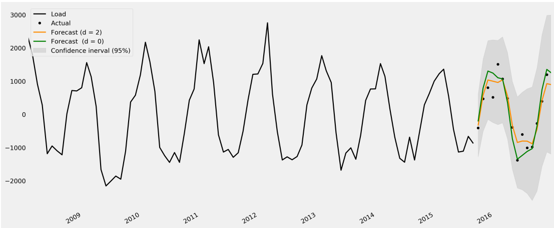


Figure 2.31: **Forcasting of the monthly avarage of Polish Electric Load for 2016.** Two different models were used: Arima(6,2,6)[Orange] and Arma(6,8)[Green].

The two models behave similarly on unexplored data, in particular the integrated model performs slightly better as it is given an RMSE of 270.22 which is approximately 9% of the global maximum of the data; The ARMA model, on the other hand, has an RMSE equal to 345.64 (12% of the maximum).

To conclude, it is interesting to note how the non-integrated model seems to be able to better capture the characteristic frequency of the data at this level, this prompted us to investigate this class of models better to verify if there was a way to better represent the time series under exam.

### Sarima modeling

The results obtained by building the ARIMA model just described have highlighted some very interesting aspects of the dataset. First of all, the fact that the autoregressive terms necessary to describe the data are a fairly high number suggests that there is a correlation between the data at least 6 lag apart. In fact, we know that the studied dataset has recurring patterns with an annual period and it might be more appropriate to build a model that takes into account the **seasonal component** to obtain accurate forecasts. For this purpose, a model called SARIMA was taken into consideration which represents a more sophisticated version of the ARMA models and introduces parameters to manage the seasonality of the dataset, generally indicated with the acronym SARIMA(p,d,q)x(P,D,Q,s) where

- **p** and seasonal **P**: indicate number of autoregressive terms (lags of the stationarized series)
- **d** and seasonal **D**: indicate differencing that must be done to stationarize series
- **q** and seasonal **Q**: indicate number of moving average terms (lags of the forecast errors)
- **s**: indicates seasonal length in the data

Retracing the steps described above, two approaches were taken to determine the ideal SARIMA parameters: ACF and PACF plots, and a grid search. Let's proceed step by step, first we used a function of statsmodel called *seasonal decompose* which allows to obtain a decomposition of the time series in an additive sense.

$$y(t) = T(t) + S(t) + N(t)$$

where:

- $y(t)$ is the time series at the time step t
- $T(t)$ is the **trend** component at the time step t

- $S(t)$ is the **seasonal** component at the time step t
- $N(t)$ is the **noise** component at the time step t

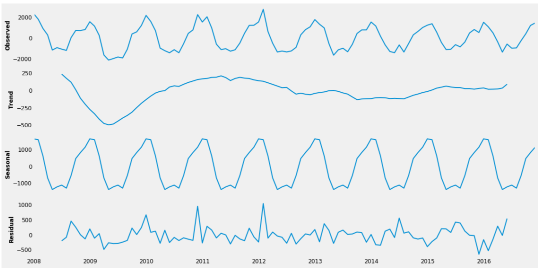

Figure 2.32: Series Additive decompositition of monthly avarage dataset.

Figure 2.32 actually shows that there is a strong seasonal component of length equal to 7 time steps. The idea now, as already mentioned, is to identify the regular (**d**) and seasonal (**D**) integration parameters and the seasonal one (**s**), then extrapolate the others through grid search. We already have information on the parameter d thanks to the previous analysis, to try to identify D we proceed by carrying out a hypothesis test AD Fuller on the seasonally differentiated series to defend seasonal lag values.

The P values are shown in figure 2.33, which shows how according to this test the best value for parameter **s** should be 6. It is important to underline that this hypothesis was subsequently confirmed by carrying out a bit of grid search also on this parameter, in fact the configurations with the lowest AIC are those that have the parameter s equal to 6. Going over again the process applied for the ARMA model, the series seems

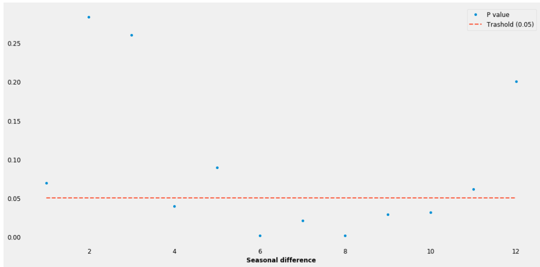

Figure 2.33: Scatter plot for P Values releted to AD-Fuller test for stationarity of differentiated series. On X axis the differencing order taken for the time series.

to assume a stationary behavior by applying a second order differentiation to the seasonally differentiated series at the first order, in short, the best configuration for the integration parameters appears to be d=2 and D=1.

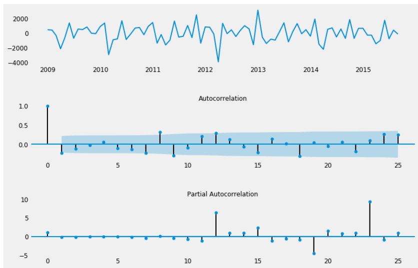

Figure 2.34: Autocorrelation and Partial Autocorrelation plots for the differentiated series of order d = 2 and D = 1.

Recalling what has been said for the ARIMA model, let's proceed by testing the behavior of both cases. The grid search on the parameters p, q, P, Q showed that the models which are in best agreement with the time series are SARIMA(3, 2, 1)x(1, 1, 2, 6) and SARIMA(3, 0, 1)x(1, 1, 2, 6). The following table shows the values of the parameters relating to both models. Once again the integrated model appears to be better according to the information criteria and diagnostic confirms the agreement of the model to the time series.

| Coeff | d=0 | d=2 |
|---|---|---|
| ar.L1 | -0.7437 | -0.7343 |
| ar.L2 | 0.6337 | -0.3098 |
| ar.L3 | . 0.3774 | 0.0616 |
| ma.L1 | 1 | 0.9992 |
| ar.S.L1 | -1 | -0.9992 |
| ma.S.L1 | -0.1438 | -0.2194 |
| ma.S.L2 | -0.8533 | -0.7738 |
| AIC | 1293.954 | 1269.353 |
| BIC | 1313.589 | 1288.799 |

Once again the integrated model appears to be better according to the information criteria and diagnostic confirms the agreement of the model to the time series.

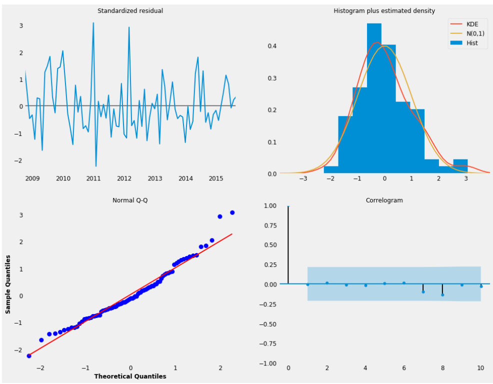


Figure 2.35: Diagnostic plot for SARIMA(3, 2, 1)x(1, 1, 2, 6) model.

At this point it remains only to check the behavior on the test set of these models. Finally, both models seem to learn better how to model the frequency that characterizes the time series, despite the fact that the error is higher than that estimated by not considering seasonality; in fact RMSE for the integrated model is equal to 498.38 which is approximately 18% of the maximum, while the other one is 332.94, approximately 12% of the maximum.

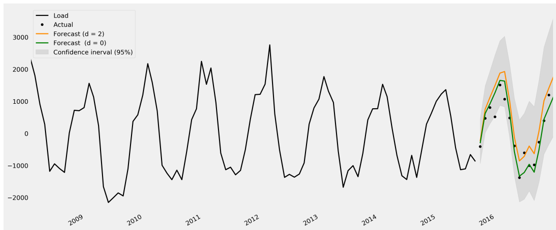


Figure 2.36: Forcasting of the monthly avarage of Polish Electric Load for 2016. Two different models were used: SARIMA(3, 2, 1)x(1, 1, 2, 6)[Orange] and SARIMA(3, 0, 1)x(1, 1, 2, 6)[Green].

## 2.6 Deconvolution

The last method that was used in the time series analysis was a deconvolution. More in detail, the idea was to hypothesize that the signal was the result of a convolution between a series of pulses and a kernel function that modeled its shape.

$$TS = d \circledast K$$

where TS represents the time series, d the pulse signal, which for convenience will be called Delta signal and K the kernel function.

The main purpose of this approach is to obtain a model capable of making predictions, to achieve this, it is necessary to know the delta signal and have an estimate of the Kernel function form.

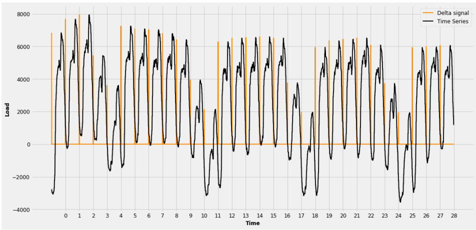

Figure 2.37: Lineplot of the first 30 days of of Polish Electric Load[Black], Representation of the pulse signal through a series of deltas centered at the starting point of each day and with an amplitude equal to the maximum relative to that day[Orange].

For simplicity, it has been assumed that the pulse signal was composed of a series of delta functions linked together with an amplitude equal to the maximum value of the time series during the day to which they refer; as regards the kernel function, it was obtained by considering an average of the time series at a daily level, in order to capture the behavior of the "average day". In fact, the figure shows how generally weekdays have a very similar trend, while holidays seem to have their own. This way we get 3 different kernels, one for weekdays, one for Saturday and one for Sunday.

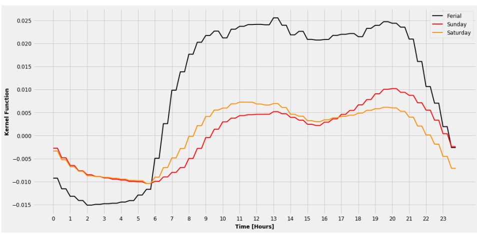

Figure 2.38: Kernel functions used for convolution.

Once defined who are the elements at stake, the model was built in the following way: The Delta signal, which represents the average year, was obtained by taking the average over all years of the maximums relating to each specific day, i.e. the amplitude of the delta relating to the first Monday of January is the average of the maximums of the first Monday of January from 2008 to 2015, the amplitude of the first Tuesday is the average of the highs of all the first Tuesdays, etc. The convolution of this delta signal with a kernel function that discriminates the weekdays from Saturday and Sunday, allows to obtain a representation of a "typical" year of the time series, thus obtaining the realization of an "average" full year. the convolution was obtained by exploiting the fourer space in which, as we know, the covolution becomes a product between the delta function and the kernel function:

$$F[TS] = F[d \circledast K] = F[d] \cdot F[K]$$

In particular we compared the model with the year 2016, which is used as a test set to see if it could actually be compared with the average behavior of the previous 8 years.

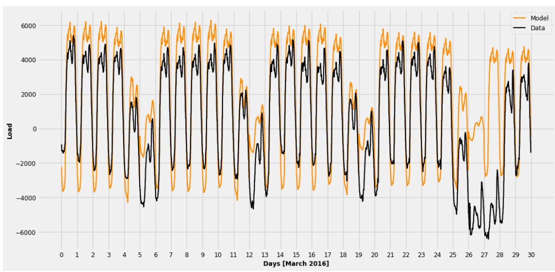

Figure 2.39: Comparison between model and time series for March 2016

Obviously this model, for how it was built is an approximation of the average behavior of the Polish electric load, figure 2.39 shows its limits; It can be seen clearly in the figure that the model manages to capture the behavior of the data, but it remains quite inaccurate. Quantitatively, The correlation between the model residuals and the time series is 0.55 and the RMSE for the data showed in 2.39 is approximately 2150 which is the 31.39% of the maximum.

# Chapter 3

# Results and Conclusion

A wide analysis has been made about the load of the Polish Power Systems using different techniques. The information that has been extracted by this analysis has been used to gain some forecasting abilities about the dataset that has been considered.
As it was predictable, the time series presents a recurrent periodical behavior. This periodicity of the data permitted to use the **Fourier Transform method**. In order to make the Fourier Analysis more robust and clean, the signal has been preprocessed in the following way:

- **De-Trend**: A linear fit has been subtracted from the signal in order to make it "more stationary".

- **De-Mean**: The mean of the signal has been subtracted from the original one in order not to disturb the frequency spectrum ($F(w = 0) = Mean = 0$)

- **Smoothing**: In order to make the endpoints of the signal meet and make the periodic assumption more valid, the hanning window has been applied to the signal

After this preprocessing part, the signal has been analyzed in the Fourier spectrum, thus retrieving life ticked frequencies like the 12h, 24h and 168h ones. The Fourier spectrum has been filtered by setting the frequencies with amplitude under a certain threshold to 0. This threshold has been chosen to vary as a certain fraction of the maximum amplitude of the original spectrum. In particular, a reconstruction algorithm that considered the correlation between the original signal and the reconstructed one permitted to obtain an optimal reconstruction by using the threshold $k_{opt} = 0.004 \cdot Max$. The same reconstruction algorithm has been applied to the first 7 years of the dataset (training set) in order to obtain the optimal frequency spectrum. **This spectrum has been used to predict the last year (test set) signal obtaining an RMSE up to the 33% of the maximum value.**
A refinement of this algorithm [4] has been used to consider the non-stationary nature of the signal. **This new approach considered a tripartite division (training/validation/test) and computed annual Fourier transforms, thus obtaining an RMSE up to the 21% of the original maximum.**
A more robust cleaning method on the signal that has been applied is the **Wavelet**

one. Under the safe assumption that the "real" signal is band-limited and the highest frequencies of the signal could be related to the noise, an algorithm has been developed in order to reconstruct a signal that could be as much as possible entirely and exclusively not influenced by the noise [3]. **This algorithm permitted to have a signal that has an error almost uncorrelated with the original signal:** $C = 0.2\%$.

Furthermore, the dataset has been studied using a class of models widely used to model time series, the ARMA models. This analysis showed how these models are able to adapt to the low frequencies of the dataset and manage to capture their behavior in a convincing way. In particular, different models were built and compared, starting from simpler models up to more sophisticated ones. It is interesting to note that in terms of RMSE the one that seems to perform better is the ARIMA(6,2,6) which does not take into account seasonality directly, but the information of the correlation between the data is mainly entrusted to the autoregressive terms. However, the two SARIMA models considered seem to be able to better capture the frequency-level behavior of the dataset by not forcibly chasing those data that could be associated with fluctuations. Probably a more precise tuning of the parameters could highlight other aspects, however this analysis has shown how also relatively simple models can be extremely powerful in this field. The last analysis performed on the time series was deconvolution, assuming that the series could be represented by the convolution of a Kernel function with a pulse function on a daily scale. The aim was to be able to create an impulse signal that represented the average annual behavior of the series, in order to use it to make predictions. Unfortunately this method did not prove to be particularly efficient, although it is possible to obtain some information; the match between the data and the model is not satisfactory for several reasons, However, we must think that there have been different degrees of approximation and the impulse signal has been constructed on the basis of a priori assumptions that do not necessarily reflect the nature of the process. Anyway looking closely at the comparison between the data and the model, it would seem that one of the problems is related to the kernel function, in fact it is possible that the average behavior of weekdays and holidays is not a good approximation of the signal behavior at the daily level because this changes consistently for each specific day. The following table shows the RMSE values related to the various methods used for forcasting the series

| Method | RMSE |
|---|---|
| Fourier | |
| Wavelet | |
| ARMA(6,8) | . 345.64 |
| ARIMA(6,2,6) | 270.22 |
| SARIMA(3, 2, 1)x(1, 1, 2, 6) | 498.38 |
| SARIMA(3, 0, 1)x(1, 1, 2, 6) | 332.94 |
| Deconvolution | 2150 |

# Chapter 4

# Appendix

In this section, the codes that permitted to obtain the ouput shown in the report have been reported. The codes are collected in their relative GitHub page. The description of the notebook is the following:

1. "datapreproccesing.ipynb"
   In this notebook data has been preprocessed: the mean value of the data and a linear fit has been subtracted by the original data. Moreover an hanning window has been applied.

2. "fouriermethod.ipynb"
   Time independent Fourier Analysis has been performed. By the usage of an opportune filter, a forecast has been made.

3. "fouriermethod2.ipynb"
   Time dependent Fourier Analysis has been performed. By the usage of an opportune filter, a forecast has been made.

4. "waveletfiltering.ipynb"
   A wavelet filter has been adopted to clean the signal.

5. "SARIMA.ipynb"
   Sarima and Arima processes have been made in order to perform a forecast on the dataset that has been cleaned by the wavelet filtering.

6. "Deconvolution.ipynb"
   A deconvolution algorithm has been applied and a forecast has been made on the last year available (2016)

## 4.1 datapreprocessing.py

```
#!/usr/bin/env python
# coding: utf-8

# # Visualization Notebook: Pre-
    processing

# In[4]:


#Importing the libraries to watch
    the 'fits' image and get the
    data array
import astropy
import plotly.graph_objects as go
from astropy.io import fits
#Importing a library that is useful
     to read the original file
import pandas as pd
import pylab as plb
import matplotlib.pyplot as plt
from scipy.optimize import
    curve_fit
```

```python
from scipy import asarray as ar,exp
#Importing a visual library with some illustrative set up
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
from matplotlib import cm
import numpy as np
import math
import seaborn as sns
plt.style.use('fivethirtyeight')
plt.rcParams['font.family'] = 'sans-serif'
plt.rcParams['font.serif'] = 'Ubuntu'
plt.rcParams['font.monospace'] = 'Ubuntu Mono'
plt.rcParams['font.size'] = 14
plt.rcParams['axes.labelsize'] = 12
plt.rcParams['axes.labelweight'] = 'bold'
plt.rcParams['axes.titlesize'] = 12
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12
plt.rcParams['legend.fontsize'] = 12
plt.rcParams['figure.titlesize'] = 12
plt.rcParams['image.cmap'] = 'jet'
plt.rcParams['image.interpolation'] = 'none'
plt.rcParams['figure.figsize'] = (16, 8)
plt.rcParams['lines.linewidth'] = 2
plt.rcParams['lines.markersize'] = 8
plt.rcParams["axes.grid"] = False

colors = ['xkcd:pale orange', 'xkcd:sea blue', 'xkcd:pale red', 'xkcd:sage green', 'xkcd:terra cotta', 'xkcd:dull purple', 'xkcd:teal', 'xkcd: goldenrod', 'xkcd:cadet blue',
'xkcd:scarlet']
cmap_big = cm.get_cmap('Spectral', 512)
cmap = mcolors.ListedColormap(cmap_big(np.linspace(0.7, 0.95, 256)))
bbox_props = dict(boxstyle="round,pad=0.3", fc=colors[0], alpha=.5)


# In[8]:


data=pd.read_csv('data.csv',sep=';')


# In[10]:


data=data.rename(columns={'Data':'Day','Godzina':'hour','Minuty':'minutes','Wolumen':'Load'})


# In[11]:


#Building a continous time array
data['seconds']=np.arange(0,len(data)*900,900)


# In[15]:


plt.plot(data.seconds,data.Load,',',color='k')
plt.grid(True)
plt.xlabel('Time (s)',fontsize=20)
plt.ylabel('Load (MW)',fontsize=20)
plt.title('Load vs Time scatterplot',fontsize=20)


# In[16]:


from scipy import signal


# In[17]:
```

```


#Detrend the signal of its constant and linear trend


# In[18]:


detrended_sig=signal.detrend(data.Load)


# In[29]:


plt.subplot(1,2,1)
plt.title('Detrended Signal',color='red',fontsize=20)
plt.plot(data.seconds,detrended_sig,',',color='red',label='Detrended')
plt.xlabel('Time (s)')
plt.ylabel('Load (MW)')
plt.subplot(1,2,2)
plt.title('Original Signal',color='k',fontsize=20)
plt.plot(data.seconds,data.Load,',',color='black',label='Original')
plt.xlabel('Time (s)')
plt.ylabel('Load (MW)')


# In[32]:


#Multiplying the signal by an hanning window

plt.plot(detrended_sig*np.hanning(len(data)),',',color='k')
plt.grid(True)
plt.xlabel('Time($s$)')
plt.ylabel('Load(MW)')
```

## 4.2 fouriermtehod.ipynb

```
#!/usr/bin/env python
# coding: utf-8

# # Time Independent Fourier Transform

# In[2]:


#Importing the libraries to watch the 'fits' image and get the data array
import astropy
import plotly.graph_objects as go
from astropy.io import fits
#Importing a library that is useful to read the original file
import pandas as pd
import pylab as plb
import matplotlib.pyplot as plt
from scipy.stats import chisquare

from scipy.optimize import curve_fit
from scipy import asarray as ar,exp
#Importing a visual library with some illustrative set up
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
from matplotlib import cm
import numpy as np
import math
from sklearn.metrics import mean_squared_error
import seaborn as sns
from scipy import signal
plt.style.use('fivethirtyeight')
plt.rcParams['font.family'] = 'sans-serif'
plt.rcParams['font.serif'] = 'Ubuntu'
plt.rcParams['font.monospace'] = 'Ubuntu Mono'
plt.rcParams['font.size'] = 14
plt.rcParams['axes.labelsize'] = 12
plt.rcParams['axes.labelweight'] = 'bold'
plt.rcParams['axes.titlesize'] = 12
plt.rcParams['xtick.labelsize'] =
```

```python
    12
plt.rcParams['ytick.labelsize'] = 12
plt.rcParams['legend.fontsize'] = 12
plt.rcParams['figure.titlesize'] = 12
plt.rcParams['image.cmap'] = 'jet'
plt.rcParams['image.interpolation'] = 'none'
plt.rcParams['figure.figsize'] = (16, 8)
plt.rcParams['lines.linewidth'] = 2
plt.rcParams['lines.markersize'] = 8
plt.rcParams["axes.grid"] = False

colors = ['xkcd:pale orange', 'xkcd:sea blue', 'xkcd:pale red', 'xkcd:sage green', 'xkcd:terra cotta', 'xkcd:dull purple', 'xkcd:teal', 'xkcd: goldenrod', 'xkcd:cadet blue',
'xkcd:scarlet']
cmap_big = cm.get_cmap('Spectral', 512)
cmap = mcolors.ListedColormap(cmap_big(np.linspace(0.7, 0.95, 256)))
bbox_props = dict(boxstyle="round,pad=0.3", fc=colors[0], alpha=.5)


# In[3]:


data=pd.read_csv('data.csv',sep=';')
data=data.rename(columns={'Data':'Day','Godzina':'hour','Minuty':'minutes','Wolumen':'Load'})
data['seconds']=np.arange(0,len(data)*900,900)
detrended_sig=signal.detrend(data.Load)
sig=detrended_sig*np.hanning(len(detrended_sig))


# In[4]:


data=data.rename(columns={'Data':'Day','Godzina':'hour','Minuty':'minutes','Wolumen':'Load'})


# In[5]:


#Building a continous time array
data['seconds']=np.arange(0,len(data)*900,900)


# In[7]:


detrended_sig=signal.detrend(data.Load)
sig=detrended_sig*np.hanning(len(detrended_sig))


# ## Uniform varying threshold

# In[8]:


#Computing the FFT
FFT=np.fft.fft(sig)


# In[9]:


#Constructing the period x axis with the hours
new_N=int(len(FFT)/2)
f_nat=1/900
new_X = np.linspace(10**-12, f_nat/2, new_N, endpoint=True)
new_Xph=1.0/(new_X*60*60)


# In[10]:
```

```
FFT_abs=np.abs(FFT)
plt.plot(new_Xph,2*FFT_abs[0:int(len(FFT)/2.)]/len(new_Xph),color='red')
plt.xlabel('Period ($h$)',fontsize=20)
plt.ylabel('Amplitude',fontsize=20)
plt.title('(Fast) Fourier Transform Method Algorithm',fontsize=20)
plt.grid(True)
plt.xlim(0,200)


# In[11]:


#Minimal difference has been shown in the reconstruction
plt.subplot(1,2,1)
plt.plot(data.seconds,np.fft.ifft(FFT),',',color='red')
plt.grid(True)
plt.xlabel('Time ($s$)')
plt.ylabel('Reconstructed Signal')
plt.subplot(1,2,2)
plt.plot(data.seconds,sig,',',color='k')
plt.grid(True)
plt.xlabel('Time ($s$)')
plt.ylabel('Original Signal')


# In[12]:


#Defining the filtering function
def fft_filter(th):
    fft_tof=FFT.copy()
    fft_tof_abs=np.abs(fft_tof)
    fft_tof_abs=2*fft_tof_abs/len(new_Xph)
    fft_tof[fft_tof_abs<=th]=0
    return fft_tof


# In[13]:


#Computing the RMSE for each reconstruction
K=np.arange(0,2475+75,75)
RMSE=[]
for k in K:
    rec_four=fft_filter(k)
    rec=np.fft.ifft(rec_four)
    RMSE.append(np.sqrt(mean_squared_error(rec.real,sig)))


# In[14]:


plt.plot(K,RMSE,color='k')
plt.xlabel('Threshold')
plt.ylabel('RMSE')
plt.grid(True)
plt.xticks(K,rotation=45)


# In[15]:


#Showing the plots at different thresholds values
#Defining the amplitude filtering function
def fft_filter_amp(th):
    fft_tof=FFT.copy()
    fft_tof_abs=np.abs(fft_tof)
    fft_tof_abs=2*fft_tof_abs/len(new_Xph)
    fft_tof_abs[fft_tof_abs<=th]=0
    return fft_tof_abs[0:int(len(fft_tof_abs)/2.)]


# In[16]:


K_plot=[10,200,700,1500]
j=0
for k in K_plot:
    j=j+1
    plt.subplot(2,2,j)
    plt.title('k=%i'%(k))
    plt.xlim(0,200)
```

```
    plt.plot(new_Xph,2*FFT_abs[0:int(len(FFT)/2.)]/len(new_Xph),color='navy',alpha=0.5,label='Original')
    plt.grid(True)
    plt.plot(new_Xph,fft_filter_amp(k),'red',label='Filtered')
    plt.xlabel('Time($h$)')
    plt.ylabel('Amplitude')
    plt.legend()
plt.subplots_adjust(hspace=0.5)


# ## Maximum related threshold

# In[17]:


#Maximum relate filter function
def fft_filter(perc):
    th=perc*(2*FFT_abs[0:int(len(FFT)/2.)]/len(new_Xph)).max()
    fft_tof=FFT.copy()
    fft_tof_abs=np.abs(fft_tof)
    fft_tof_abs=2*fft_tof_abs/len(new_Xph)
    fft_tof[fft_tof_abs<=th]=0
    return fft_tof


# In[18]:


#Showing some plots at different threshold values
K_plot_values=[0.0,0.30,0.60,0.95]
j=0
for k in K_plot_values:
    j+=1
    plt.subplot(4,1,j)
    plt.plot(data.seconds,np.fft.ifft(fft_filter(k)),color=colors[j])
    plt.title('k=%.2f of the maximum' %(k))
    plt.xlabel('Time ($s$)')
    plt.ylabel('Load')
plt.subplots_adjust(hspace=0.8)


# In[19]:


#Performing the same RMSE process as before, but relating to the maximum value
#Computing the RMSE for each reconstruction
K=np.arange(0.0,0.31,0.01)
RMSE=[]
for k in K:
    rec_four=fft_filter(k)
    rec=np.fft.ifft(rec_four)
    RMSE.append(np.sqrt(mean_squared_error(rec.real,sig)))


# In[20]:


plt.plot(K,RMSE,'.',color='k')
plt.grid(True)
plt.xlabel('Threshold as percentage of the maximum value ',fontsize=20)
plt.xticks(K,rotation=90)
plt.ylabel('RMSE',fontsize=20,rotation=90)


# In[21]:


#Computing the correlation between the original signal and its error
CORR=[]
for k in K:
    rec_four=fft_filter(k)
    rec=np.fft.ifft(rec_four)
    error=rec.real-sig
    CORR.append(np.abs(np.corrcoef(sig,error)[0][1]))


# In[22]:

```

```python

#Plotting the correlation between the signal and its error
plt.plot(K,CORR,'.',color='k')
plt.grid(True)
plt.xlabel('Threshold as percentage of the maximum value ',fontsize=20)
plt.xticks(K,rotation=90)
plt.ylabel('Correlation Coefficient',fontsize=20,rotation=90)


# In[23]:


#Reducing the range and selecting the best K value
K=np.arange(0.001,0.015,0.001)
CORR=[]
for k in K:
    rec_four=fft_filter(k)
    rec=np.fft.ifft(rec_four)
    error=rec.real-sig
    CORR.append(np.abs(np.corrcoef(sig,error)[0][1]))
plt.plot(K,CORR,'.',color='k')
plt.plot(K,np.zeros(len(K))+0.10,color='red')
plt.grid(True)
plt.xlabel('Threshold as percentage of the maximum value ',fontsize=20)
plt.xticks(K,rotation=90)
plt.ylabel('Correlation Coefficient',fontsize=20,rotation=90)


# In[24]:


#Displaying the optimum values
opt_perc=0.004
plt.subplot(2,2,1)
plt.plot(data.seconds,np.fft.ifft(fft_filter(opt_perc).real),',',color='red')
plt.grid(True)
plt.xlabel('Time($s$)')
plt.ylabel('Reconstructed Signal')
plt.subplot(2,2,2)
plt.plot(data.seconds,sig,',',color='k')
plt.grid(True)
plt.xlabel('Time($s$)')
plt.ylabel('Original Signal')
plt.subplot(2,2,4)
plt.xlim(0,200)
plt.plot(new_Xph,2*FFT_abs[0:int(len(FFT)/2.)]/len(new_Xph),color='k',alpha=1.0,label='Original')
plt.legend()

plt.grid(True)
plt.xlabel('Time($h$)')
plt.ylabel('Original Fourier Transform amplitude')
plt.subplot(2,2,3)
plt.xlim(0,200)
plt.plot(new_Xph,2*np.abs(fft_filter(opt_perc))[0:int(len(FFT)/2.)]/len(new_Xph),color='red',alpha=1.0,label='Reconstructed')
plt.grid(True)
plt.xlabel('Time($h$)')
plt.ylabel('Reconstructed Fourier Transform amplitude ')
plt.legend()


# In[25]:


#Distribution of the error
sns.distplot(sig-np.fft.ifft(fft_filter(opt_perc)).real,color='firebrick')
plt.grid(True)
plt.xlabel('Error distribution values',fontsize=20)
plt.ylabel('Distribution',fontsize=20)


# In[26]:

```

```python

y_1=np.histogram(sig-np.fft.ifft(fft_filter(opt_perc)).real,500)[0]
x_1=np.histogram(sig-np.fft.ifft(fft_filter(opt_perc)).real,500)[1][0:len(y_1)]


# In[27]:



def gaus(x,a,x0,sigma):
        return a*np.exp(-(x-x0)**2/(2*sigma**2))


# In[28]:


#x_1=x_1[np.where((x_1>-500) & (x_1<500))]
y_1=y_1[np.where((x_1>-500) & (x_1<500))]
x_1=np.linspace(-500,500,len(y_1))


# In[29]:


#Plot of the values
val_medio=0
n = len(x_1)                          #the number of data
mean = sum(x_1*y_1)/n                  #note this correction
sigma = sum(y_1*(x_1-val_medio)**2)/n        #note this correction
p0 = [max(y_1),val_medio,10]
popt,pcov = curve_fit(gaus,x_1,y_1,p0=p0)
fig, (ax_11, ax_12) = plt.subplots(2, 1)
plt.suptitle('Fit and rediduals', fontsize=20)
#ax_11.set_y_1_label('Intensity_1 [ADU]')
ax_11.plot(x_1,y_1,'navy',label='Distribution')
ax_11.grid(True)
ax_11.set_xlabel('Distribution Values')
ax_11.set_ylabel('Distribution')

#plt.y_1label('Residuals')
ax_11.plot(x_1,gaus(x_1,*popt),'red',label='fit')
plt.grid(True)
ax_11.legend()
res = y_1 - gaus(x_1,*popt)
ax_12.plot(x_1,res,color='darkorange',label='Residuals')
ax_11.set_xlabel('Distribution Values')
ax_11.set_ylabel('Residuals')
ax_12.legend()
plt.show()


# ## Prediction Part

# # Train Test Split

# In[30]:


#Selecting the first 80% of the signal
sig=data.Load
sig=signal.detrend(sig)
sig=sig[0:365*96*8].copy()
FFT=np.fft.fft(sig)
FFT_abs=np.abs(FFT)


# In[31]:


#Maximum relate filter function
def fft_filter(perc):
    sig=data.Load
    sig=signal.detrend(sig)
    sig=sig[0:365*96*8].copy()
    FFT=np.fft.fft(sig)
    FFT_abs=np.abs(FFT)
    th=perc*(2*FFT_abs[0:int(len(FFT)/2.)]/len(new_Xph)).max()
```

```
    fft_tof=FFT.copy()
    fft_tof_abs=np.abs(fft_tof)
    fft_tof_abs=2*fft_tof_abs/len(new_Xph)
    fft_tof[fft_tof_abs<=th]=0
    return fft_tof


# In[32]:


#Performing the same RMSE process as before, but relating to the maximum value
#Computing the RMSE for each reconstruction
K=np.arange(0.001,0.011,0.001)
RMSE=[]
CORR=[]
for k in K:
    rec_four=fft_filter(k)
    rec=np.fft.ifft(rec_four)
    RMSE.append(np.sqrt(mean_squared_error(rec.real,sig)))
    rec=np.fft.ifft(rec_four)
    error=rec.real-sig
    CORR.append(np.abs(np.corrcoef(sig,error)[0][1]))


# In[33]:


plt.subplot(1,2,1)
plt.plot(K,RMSE,'.',color='k')
plt.grid(True)
plt.xlabel('Threshold as percentage of the maximum value ',fontsize=20)
plt.xticks(K,rotation=90)
plt.ylabel('RMSE',fontsize=20,rotation=90)
plt.subplot(1,2,2)
plt.plot(K,CORR,'.',color='k')
plt.grid(True)
plt.xlabel('Threshold as percentage of the maximum value ',fontsize=20)
plt.plot(K,np.zeros(len(K))+0.10,color='red')
plt.xticks(K,rotation=90)
plt.ylabel('Correlation',fontsize=20,rotation=90)


# In[34]:


opt_perc=0.004
fft_filter(opt_perc)
plt.xlim(0,200)
plt.ylim(0,2500)
plt.plot(new_Xph,np.abs(fft_filter(opt_perc))[0:len(new_Xph)]/len(new_Xph),color='k',alpha=1.,label='Original')
plt.xlabel('Time $h$')
plt.ylabel('Amplitude')
plt.grid(True)
plt.title('Fourier Spectrum',fontsize=20)


# In[203]:


#Prediction:
#Using the Fourier formula
fourier=fft_filter(opt_perc)
space=np.array(data.seconds.tolist())
f_nat=1/(space[1]-space[0])
P=np.pi*2
#N=int(len(space)/2)
freq=np.linspace(0,f_nat/2,len(fourier))*2*np.pi
#fourier=f_try[1]
J_LIST=np.where(fourier!=0)[0]
real=fourier.real
imag=fourier.imag
T=np.linspace(0,f_nat/2,len(freq),endpoint=True)
I=np.arange(1,101,1)
x_t=np.zeros(len(space))
q=0
SEEN=[]
for j in J_LIST:
```

```
     q=q+1
     x_t=x_t+2/(len(space))*(np.abs(
    fourier[j].real)*np.cos(freq[j
    ]*2*space)-fourier[j].imag*np.
    sin(2*freq[j]*space))
#print(i,len(space))
     if int(100*q/len(J_LIST)) in I
    and int(100*q/len(J_LIST)) not
    in SEEN:
          print('%i'%(int(100*q/len(
    J_LIST)))+ ' % of the
    frequencies reconstructed')
          SEEN.append(int(100*q/len(
    J_LIST)))
     #X_T.append(x_t)


# In[200]:


x_t=np.array(x_t)
x_t[np.where(x_t>sig.max())]=sig.
    max()


# In[312]:


#plt.plot(np.fft.ifft(fourier))
sig=data.Load
sig=signal.detrend(sig)
plt.plot(space,sig,color='k')
plt.plot(space[0:len(fourier)],x_t
    [0:len(fourier)],color='
    darkorange')
plt.plot(space[len(fourier):len(
    data)],x_t[len(fourier):len(
    data)],color='gold')
plt.grid(True)
plt.xlabel('Time($s$)')
plt.ylabel('Load (MW)')
#plt.plot(space[len(fourier)+10:len
    (data)],exp,color='black')
#plt.plot(data.seconds[0:365*8*96],
    np.fft.ifft(fourier))
##plt.plot(data.seconds
    [365*8*96::],sig[365*8*96::])
#plt.plot(space,x_t*max(sig)/max(
    x_t))


# In[307]:


RMSE=mean_squared_error(sig
    [365*8*96:len(data)],np.array(
    x_t[365*8*96:len(data)]).real)


# In[309]:


print('RMSE is '+ str(RMSE))


# In[314]:


plt.plot(data.seconds[365*8*96:len(
    data)],sig[365*8*96:len(data)],
    '.',color='gold',label='
    Prediction',markersize=5)
plt.plot(data.seconds[365*8*96:len(
    data)],x_t[365*8*96:len(data)],
    '.',color='k',label='Real',
    markersize=5)
plt.legend()
plt.grid(True)
plt.xlabel('Time($s$)')
plt.ylabel('Signal')


# In[321]:


#Plotting the errors
plt.plot(np.array(data.seconds)
    [0:365*8*96],(sig-x_t)
    [0:365*8*96],color='k')
plt.plot(np.array(data.seconds)
    [365*8*96:len(data)],(sig-x_t)
    [365*8*96:len(data)],color='
    gold')
plt.xlabel('Time($s$)')
plt.ylabel('Load($MW$)')
plt.grid(True)
```

```python
# In[352]:


#RMSE for each interval

D=[0,20,30,60,90,120]
D=np.array(D)*900
start=365*900*8
T_in=(D+start)[:-1]
T_fin=(D+start)[1::]
D=[0,20,30,60,90,120]
new_D=np.array(D)
new_D=new_D*96
new_start=365*96*8
RMSE=[]
C=[]
for d in range(len(D)-1):
    x_t_=np.array(x_t[(new_start+new_D[d]):(new_start+new_D[d+1])])
    sig_=sig[(new_start+new_D[d]):(new_start+new_D[d+1])]
    error=x_t_.real-sig_
    RMSE.append(np.sqrt(mean_squared_error(sig_,x_t_.real)))
    C.append(np.corrcoef(error,sig_)[0][1])


# In[357]:


RMSE_data=pd.DataFrame()
RMSE_data['$T_fin$(s)']=T_fin
RMSE_data['RMSE']=RMSE
RMSE_data['$\Delta T$(s)']=T_fin-T_in
RMSE_data['$C_{data,error}$']=C
RMSE_data['$\Delta Days$']=D[1::]


# In[360]:


RMSE_data
```

## 4.3 fouriermethod2.ipynb

```python
#!/usr/bin/env python
# coding: utf-8

# # Time Dependent method

# In[2]:


#Importing the libraries to watch the 'fits' image and get the data array
import astropy
import plotly.graph_objects as go
from astropy.io import fits
#Importing a library that is useful to read the original file
import pandas as pd
import pylab as plb
import matplotlib.pyplot as plt
from scipy.stats import chisquare

from scipy.optimize import curve_fit
from scipy import asarray as ar,exp
#Importing a visual library with some illustrative set up
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
from matplotlib import cm
import numpy as np
import math
from sklearn.metrics import mean_squared_error
import seaborn as sns
from scipy import signal
plt.style.use('fivethirtyeight')
plt.rcParams['font.family'] = 'sans-serif'
plt.rcParams['font.serif'] = 'Ubuntu'
plt.rcParams['font.monospace'] = 'Ubuntu Mono'
plt.rcParams['font.size'] = 14
plt.rcParams['axes.labelsize'] = 12
plt.rcParams['axes.labelweight'] = 'bold'
plt.rcParams['axes.titlesize'] = 12
```

```python
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12
plt.rcParams['legend.fontsize'] = 12
plt.rcParams['figure.titlesize'] = 12
plt.rcParams['image.cmap'] = 'jet'
plt.rcParams['image.interpolation'] = 'none'
plt.rcParams['figure.figsize'] = (16, 8)
plt.rcParams['lines.linewidth'] = 2
plt.rcParams['lines.markersize'] = 8
plt.rcParams["axes.grid"] = True
#plt.rcParams['']
colors = ['xkcd:pale orange', 'xkcd:sea blue', 'xkcd:pale red', 'xkcd:sage green', 'xkcd:terra cotta', 'xkcd:dull purple', 'xkcd:teal', 'xkcd: goldenrod', 'xkcd:cadet blue',
'xkcd:scarlet']
cmap_big = cm.get_cmap('Spectral', 512)
cmap = mcolors.ListedColormap(cmap_big(np.linspace(0.7, 0.95, 256)))
bbox_props = dict(boxstyle="round, pad=0.3", fc=colors[0], alpha=.5)


# In[3]:


data=pd.read_csv('data.csv',sep=';')
data=data.rename(columns={'Data':'Day','Godzina':'hour','Minuty':'minutes','Wolumen':'Load'})
data['seconds']=np.arange(0,len(data)*900,900)
detrended_sig=signal.detrend(data.Load)
sig=detrended_sig*np.hanning(len(detrended_sig))
```

```python


# In[4]:


load_no_line=signal.detrend(data.Load,type='linear')
clean_load=np.array(load_no_line)-np.array(load_no_line).mean()
data['clean_load']=clean_load
data=data.drop(columns=['Load']).rename(columns={'clean_load':'Load'})


# In[5]:


#Mean FFT on Training set

data.Day=pd.to_datetime(data.Day)
data['Year']=data.Day.dt.year #Years
data['Month']=data.Day.dt.month #Months
YEARS=np.sort(list(set(data.Year.tolist()))).tolist() #Year list
load_yf=0
for y in range(len(YEARS)-2): #len of the training set
    df_year=data[data['Year']==YEARS[y]]
    if y==0 or y==4: #leap years :)
        df_year=df_year.drop(df_year[df_year.Day==(str(YEARS[y])+'-02-29')].index)
    load_yf=load_yf+np.fft.fft(df_year.Load)
four_year=load_yf/(len(YEARS)-2)


# In[6]:


#Fourier transform is defined on the frequency axis.
#In order to make it more visible, the conversion on the period has been applied
```

```
new_N=int(len(four_year)/2)
f_nat=1/900
new_X = np.linspace(10**-12, f_nat/2, new_N, endpoint=True)
new_Xph=1.0/(new_X*60*60)
plt.xlim(0,40)
plt.plot(new_Xph,np.abs(2*four_year[:int(len(four_year)/2.)]/len(four_year)),color='red')
plt.ylabel('Amplitude')
plt.xlabel('Period ($h$)')
plt.ylim(0,3000)
plt.title('Mean Fourier Spectrum',fontsize=20)


# In[7]:


#Validation
TH=np.arange(0.01,0.5,0.01) # threshold Values
opt_t=[] #Threshold values that are acceptable in terms of RMSE and Correlation values
opt_corr=[] #Acceptable Correlation values
abs_year=np.array(abs(four_year))
for t in TH:
    t_abs_year=abs_year.copy()
    t_abs_year[t_abs_year<(t*t_abs_year.max())]=0 #Apply the threshold on the absolute value
    t_four_year=four_year.copy()
    t_four_year[t_abs_year==0]=0 # Apply the threshold on Fourier transform
    RMSE=np.sqrt(mean_squared_error(np.fft.ifft(t_four_year).real,data[data.Year==2015].Load)) # inverse transform and compute RMSE
    if RMSE<2000: #First requirement: LOW RMSE
        new_TH=np.arange(t*0.1,t,t*0.1) #More specific threshold
        for t_new in new_TH:
            t_new_abs_year=abs_year.copy()
            t_new_abs_year[t_new_abs_year<(t_new*t_new_abs_year.max())]=0
            t_new_four_year=four_year.copy()
            t_new_four_year[t_new_abs_year==0]=0
            E=np.fft.ifft(t_new_four_year).real-df_year.Load
            corr=pd.DataFrame({'A':df_year.Load,'B':E}).corr().values[0][1]
            if abs(corr)<=0.54: # Second requirement: LOW CORRELATION
                opt_t.append(t_new)
                opt_corr.append(corr)
opt_corr=np.abs(opt_corr)


# In[8]:


#Selecting the best reconstruction
best_th=opt_t[opt_corr.argmin()] # Best threshold value
abs_year=np.array(abs(four_year))
t_abs_year=abs_year.copy()
t_abs_year[t_abs_year<(best_th*t_abs_year.max())]=0
med_four=four_year.copy()
med_four[t_abs_year==0]=0 #Best fourier transform


# In[9]:


RMSEs=[]
test=data[data['Year']==2016] #Test year
test=test.drop(test[test.Day==('2016-02-29')].index) #Again, leap year
C=[] #Correlation list
TIME=np.arange(0,100,10) #Timestep
```

```
 we want to check
rmse_four=np.fft.ifft(med_four) # Take our prediction
test=np.array(test.Load)

for t in range(len(TIME)-1):
    Rmse_test=test[96*(TIME[0]):96*(TIME[t+1])] #Take the load between two timestep
    Rmse_four=rmse_four[96*(TIME[0]):(96*(TIME[t+1]))] #Take the prediction between the same timestep
    RMSEs.append(np.sqrt(mean_squared_error(Rmse_four.real,Rmse_test))) #Compute the RMSE
    C.append(np.corrcoef(Rmse_four.real-Rmse_test,Rmse_test)[0][1]) #Compute the correlation
res_data=pd.DataFrame({'RMSE':RMSEs, 'Days':TIME[1:], '$C_{data,error}$': C}) #Store them in a dataframe
res_data #Et voil


# In[11]:


print('Percentage of the maximum: ' + str(100*np.array(RMSEs).min()/data.Load.max())+ '%')


# In[17]:


plt.plot(rmse_four[0:30*96],color='k')
plt.plot(test[0:30*96],color='darkorange')
plt.xlabel('Time(Day)')
plt.ylabel('Load')
plt.xticks(np.arange(0,30*96,96),np.arange(0,30))
```

## 4.4 waveletfiltering.ipynb

```
#!/usr/bin/env python
# coding: utf-8

# # Wavelet Filtering

# In[4]:


#Importing the libraries to watch the 'fits' image and get the data array
import astropy
from astropy.io import fits
#Importing a library that is useful to read the original file
import pandas as pd
import pylab as plb
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
import pywt
from scipy.stats import chisquare
from scipy import asarray as ar,exp
#Importing a visual library with
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
import random
from matplotlib import cm
import matplotlib.patches as mpatches

import numpy as np
import math
import seaborn as sns
import datetime
plt.style.use('fivethirtyeight')
plt.rcParams['font.family'] = 'sans-serif'
plt.rcParams['font.serif'] = 'Ubuntu'
plt.rcParams['font.monospace'] = 'Ubuntu Mono'
plt.rcParams['font.size'] = 14
plt.rcParams['axes.labelsize'] = 12
plt.rcParams['axes.labelweight'] =
```

```
    'bold'
plt.rcParams['axes.titlesize'] = 12
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12
plt.rcParams['legend.fontsize'] = 12
plt.rcParams['figure.titlesize'] = 12
plt.rcParams['image.cmap'] = 'jet'
plt.rcParams['image.interpolation'] = 'none'
plt.rcParams['figure.figsize'] = (16, 8)
plt.rcParams['lines.linewidth'] = 2
plt.rcParams['lines.markersize'] = 8
plt.rcParams["axes.grid"] = False

colors = ['xkcd:pale orange', 'xkcd:sea blue', 'xkcd:pale red', 'xkcd:sage green', 'xkcd:terra cotta', 'xkcd:dull purple', 'xkcd:teal', 'xkcd: goldenrod', 'xkcd:cadet blue',
'xkcd:scarlet']
cmap_big = cm.get_cmap('Spectral', 512)
cmap = mcolors.ListedColormap(cmap_big(np.linspace(0.7, 0.95, 256)))
bbox_props = dict(boxstyle="round,pad=0.3", fc=colors[0], alpha=.5)


# # Pre-processing steps

# In[5]:


data=pd.read_csv('data.csv',sep=';')


# In[6]:


data=data.rename(columns={'Data':'Day','Godzina':'hour','Minuty':'minute','Wolumen':'Load'})


# In[7]:


SECONDS=np.arange(900,900*len(data)+900,900)


# In[8]:


data['seconds']=SECONDS


# In[9]:


LOAD=data.Load


# In[10]:


from scipy import signal
load_no_line=signal.detrend(LOAD, type='linear')
#load_no_constant=signal.detrend(load_no_line,type='constant')


# In[11]:


clean_load=np.array(load_no_line)-np.array(load_no_line).mean()


# In[12]:


data['clean_load']=clean_load


# In[13]:

```

```
data=data.drop(columns=['Load']).rename(columns={'clean_load':'Load'})


# # Wavelet Transform

# In[14]:


#Performing the Wavelet transform using a sym2
week=data
time=week.seconds.max()
sample_rate=1/900.
size= int(sample_rate*time)
t = np.linspace(0, time, num=size)
dataset = np.array(week.Load.tolist())
waveletname = 'sym2'
levels=8
fig, axarr = plt.subplots(nrows=levels, ncols=2, figsize=(20,10))
COEFF_D=[]
DATASET=[]
k=1
for ii in range(levels):
    (dataset, coeff_d) = pywt.dwt(dataset, waveletname,mode='per')
    axarr[ii, 0].plot(dataset, 'black')
    axarr[ii, 1].plot(coeff_d, 'darkorange')
    axarr[ii, 0].set_ylabel("Level {}".format(ii + 1), fontsize=14, rotation=90)
    axarr[ii, 0].set_yticklabels([])
    if ii == 0:
        axarr[ii, 0].set_title("Approximation coefficients", fontsize=14)
        axarr[ii, 1].set_title("Detail coefficients", fontsize=14)
    axarr[ii, 1].set_yticklabels([])
    #print(len(coeff_d))
    COEFF_D.append(np.repeat(coeff_d,2**k))
    DATASET.append(np.repeat(dataset,2**k))
    k=k+1
plt.tight_layout()
plt.show()


# In[15]:


#Preparing the gaussian fit
def gaus(x,a,x0,sigma):
        return a*np.exp(-(x-x0)**2/(2*sigma**2))




# In[16]:


#Histogram of the first coefficient
x=np.histogram(COEFF_D[0],500)[1][0:500]
y=np.histogram(COEFF_D[0],500)[0]


# In[17]:


#To restrict the number of values and fit the gaussian correctly, this function has been used
def takeClosest(num,collection):
    collection=collection.tolist()
    collection=np.array(collection)
    if num>=0:
        collection=np.abs(collection[np.where(collection>0)])
        a= min(collection,key=lambda x:abs(x-abs(num)))
    else:
        collection=np.abs(collection[np.where(collection
```

```
    <0)])

        a= -min(collection,key=lambda x:abs(x-abs(num)))
    return a


# In[18]:


#Gaussian Fit
x_try=x
b=x.tolist().index(takeClosest(500,x))
a=x.tolist().index(takeClosest(-500,x))
x=x[a:b]
y=y[a:b]


val_medio=x[int(len(x)/2)]
n = len(x)                          #the number of data
mean = sum(x*y)/n                   #note this correction
sigma = sum(y*(x-val_medio)**2)/n   #note this correction
p0 = [max(y),val_medio,10]
popt,pcov = curve_fit(gaus,x,y,p0=p0)
fig, (ax1, ax2) = plt.subplots(2, 1)
ax1.set_ylabel('Distribution ')
ax1.plot(x,y,'navy',label='First Coefficient Distribution')
ax1.grid(True)
plt.ylabel('Residuals')
ax1.plot(x,gaus(x,*popt),'ro:',label='Gaussian Fit')
ax1.legend()
res = y - gaus(x,*popt)
ax2.plot(res,color='darkorange')
ax2.grid(True)
plt.show()


# In[19]:


x=sns.distplot(COEFF_D[0],color='k').get_lines()[0].get_data()[0][a:b]
y=sns.distplot(COEFF_D[0],color='k').get_lines()[0].get_data()[1][a:b]
plt.ylabel('Distribution')
plt.xlabel('Distribution Values of the First Level')
plt.grid(True)


# In[20]:


Sigma=popt[2]


# In[36]:


#Defining the reconstruction with the threshold
def recons_from_th_zero(threshold):
    lim_1=-threshold*Sigma
    lim_2=-lim_1
    coeff_0=np.array(COEFF_D[0].copy())
    #mask = np.where((coeff_0<lim_2) & (coeff_0>lim_1))
    coeff_0[np.where((coeff_0<lim_2) & (coeff_0>lim_1))]=0
    level_0=coeff_0+DATASET[0]
    MAX=level_0.max()
    return level_0*(data.Load.max()/MAX)


# In[24]:


#Data test
test=np.array(data.Load)


# In[25]:


#Threshold, large range
```

```
TH=np.arange(0.0,13.25,0.5)
C_TH=[]
for t in TH:
    rec=recons_from_th_zero(t)
    C_TH.append(np.abs(np.corrcoef(
    rec-test,test)[0][1]))


# In[29]:


#Threshold, narrow range
TH=np.arange(0.0,1.01,0.01)
C_TH=[]
for t in TH:
    rec=recons_from_th_zero(t)
    C_TH.append(np.abs(np.corrcoef(
    rec-test,test)[0][1]))


# In[37]:


#Best threshold
print('The Correlation has the
    following percentage: '+str
    (100*C_TH[-1]) +'%')


# In[53]:


#Mean and extreme value
mean=popt[1]
lim_neg=mean-1*Sigma
lim_pos=mean+1*Sigma


# In[78]:


lim_1=-1*Sigma
lim_2=-lim_1
coeff_0=np.array(COEFF_D[0].copy())
#mask = np.where((coeff_0<lim_2) &
    (coeff_0>lim_1))
#coeff_0[np.where((coeff_0<lim_2) &
    (coeff_0>lim_1))]=0


# In[101]:


x_hist=np.histogram(coeff_0,199)[1]
y_hist=np.histogram(coeff_0,200)[0]
filt_coeff_0=coeff_0[np.where((
    coeff_0<lim_2) & (coeff_0>lim_1
    ))]


# In[104]:


x_left=x_hist[np.where((x_hist<
    lim_neg) & (x_hist>-500))]
x_right=x_hist[np.where(x_hist>
    lim_pos)]
y_left=y_hist[np.where((x_hist<
    lim_neg) & (x_hist>-500))]
y_right=y_hist[np.where(x_hist>
    lim_pos)]
x_mid=x_hist[np.where((x_hist>
    lim_neg) & (x_hist<lim_pos))]
y_mid=y_hist[np.where((x_hist>
    lim_neg) & (x_hist<lim_pos))]



# In[111]:


plt.plot(x_left,y_left,color='k')
plt.plot(x_right,y_right,color='k',
    label='Information')
plt.plot(x_mid,y_mid,color='
    darkorange',label='Noise')
plt.plot(np.zeros(12000)+lim_neg,np
    .arange(0,12000),color='red')
plt.plot(np.zeros(12000)+lim_pos,np
    .arange(0,12000),color='red')

plt.grid(True)
plt.xlabel('Coefficient values')
plt.ylabel('Coefficient
    Distribution')
plt.legend(fontsize=20)
```

## 4.5 SARIMA.ipynb

```
#!/usr/bin/env python
# coding: utf-8

# # SARIMA MODELING

# Import dataset

# In[2]:


#Importing the libraries to watch the 'fits' image and get the data array
import astropy
import plotly.graph_objects as go
from astropy.io import fits
#Importing a library that is useful to read the original file
import pandas as pd
import pylab as plb
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
from scipy import asarray as ar,exp
#Importing a visual library with some illustrative set up
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
from matplotlib import cm
import numpy as np
import math
import seaborn as sns
plt.style.use('fivethirtyeight')
plt.rcParams['font.family'] = 'sans-serif'
plt.rcParams['font.serif'] = 'Ubuntu'
plt.rcParams['font.monospace'] = 'Ubuntu Mono'
plt.rcParams['font.size'] = 14
plt.rcParams['axes.labelsize'] = 12
plt.rcParams['axes.labelweight'] = 'bold'
plt.rcParams['axes.titlesize'] = 12
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12
plt.rcParams['legend.fontsize'] = 12
plt.rcParams['figure.titlesize'] = 12
plt.rcParams['image.cmap'] = 'jet'
plt.rcParams['image.interpolation'] = 'none'
plt.rcParams['figure.figsize'] = (16, 8)
plt.rcParams['lines.linewidth'] = 2
plt.rcParams['lines.markersize'] = 8
plt.rcParams["axes.grid"] = False

colors = ['xkcd:pale orange', 'xkcd:sea blue', 'xkcd:pale red', 'xkcd:sage green', 'xkcd:terra cotta', 'xkcd:dull purple', 'xkcd:teal', 'xkcd: goldenrod', 'xkcd:cadet blue',
'xkcd:scarlet']
cmap_big = cm.get_cmap('Spectral', 512)
cmap = mcolors.ListedColormap(cmap_big(np.linspace(0.7, 0.95, 256)))
bbox_props = dict(boxstyle="round,pad=0.3", fc=colors[0], alpha=.5)


# In[3]:


from statsmodels.graphics.tsaplots import plot_pacf
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.tsa.arima_process import ArmaProcess
from statsmodels.stats.diagnostic import acorr_ljungbox
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.tsa.stattools import adfuller
from statsmodels.tsa.stattools import pacf
```

```python
from statsmodels.tsa.stattools import acf
from tqdm import tqdm_notebook
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

from itertools import product


# In[4]:


dataset = pd.read_csv('/Users/Simone/Desktop/University/DDA/Lab/Relazione_2_TimesSeries/reconstruction.csv')

dataset = dataset.drop(columns=['Unnamed: 0'])

full_load = pd.Series(dataset.R)


# In[5]:


import csv
import datetime as dat
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

dir='/Users/Simone/Desktop/University/DDA/Lab/Relazione_2_TimesSeries/'
file='daneOkresoweKSE.csv'

# Read data from file 'filename.csv'
# (in the same directory that your python process is based)
# Control delimiters, rows, column names with read_csv (see later)
data = pd.read_csv(dir+file)
# Preview the first 5 lines of the loaded data
data.head()


data = pd.read_csv(dir+file,skiprows=0,sep=';')
data.head()


data.columns= ['Day','hour','minute','Load']
#plt.figure(figsize=(10, 10))
#plt.plot(F606,F814,',k')
#plt.ylim(32,12)
#plt.xlim(32,12)
#plt.show()
#plt.close


# In[6]:


N= 10000# data.shape[0]
print("Using {} entries on {} available".format(N, data.shape[0]) )
load=np.zeros(N) #vogliamo mettere i vettori tempo in formato np, per fare successive analisi, in formato datetime pu  essere scomodo
time=np.zeros(N)


txt0 = "{} {:02d}:{:02d}:00".format(data['Day'][0],data['hour'][0],0)
T0= dat.datetime.strptime(txt0,'%Y-%m-%d %H:%M:%S')

dday=0
for i in range(N):
    load[i]=data['Load'][i]

    if data['hour'][i]==24:
        data['hour'][i]=0
        dday=1

    txt = "{} {:02d}:{:02d}:00".format(data['Day'][i],data['hour'][i],data['minute'][i])
    date= dat.datetime.strptime(txt
```

```
    ,'%Y-%m-%d %H:%M:%S')
     #Serve questo loop perch  i polacchi segnano la mezzanotte in maniera differente
     if dday==1:
         date=date+dat.timedelta(days=1)
     Secs=(date-T0).total_seconds()
     time[i]=Secs
     dday=0

plt.figure(figsize=(10,10))
plt.plot(time,load,'.k')

plt.xlabel('Time [s]')
plt.ylabel('Load [MW]')
plt.title('Polish Electric Load since {} {:02d}:{:02d}:00'.format(data['Day'][0],data['hour'][0],0) )
plt.show()
#


# In[8]:


s = data.Day.loc[0]
data.iloc[96]


# In[9]:


data.tail()


# In[10]:


from scipy import signal
dir='/Users/Simone/Desktop/University/DDA/Lab/Relazione_2_TimesSeries/'
file='daneOkresoweKSE.csv'

# Read data from file 'filename.csv'
# (in the same directory that your python process is based)
# Control delimiters, rows, column names with read_csv (see later)
data = pd.read_csv(dir+file, sep =';')
data=data.rename(columns={'Data':'Day','Godzina':'hour','Minuty':'minute','Wolumen':'Load'})
SECONDS=np.arange(900,900*len(data)+900,900)
data['seconds']=SECONDS
LOAD=data.Load
load_no_line=signal.detrend(LOAD,type='linear')
clean_load=np.array(load_no_line)-np.array(load_no_line).mean()
data['clean_load']=clean_load
data=data.drop(columns=['Load']).rename(columns={'clean_load':'Load'})


# In[11]:


s = '2008-01-01 00:15:00'
end = '2017-01-01 00:00:00'


# In[12]:


index = pd.date_range(start = s  , end = end , freq='15T')
series = pd.Series(data.Load.tolist(), index=index)
series.resample('1W').mean()


# In[13]:


W=pd.DataFrame({'Time':series.resample('1W').mean().index,'Load':series.resample('1W').mean()})
M=pd.DataFrame({'Time':series.resample('1M').mean().index,'Load':series.resample('1M').
```

```
    mean()})
D=pd.DataFrame({'Time':series.
    resample('1D').mean().index,'
    Load':series.resample('1D').
    mean()})


# In[14]:


M = M.drop(M.index[-1])


# In[15]:


plt.plot(M.Load,color = 'k')
plt.xlabel('Year')
plt.ylabel('Load')
plt.title('Monthly avarage of
    Polish Electric Load from 2008
    to 2017')


# # Stationarity

# Let's check stationary properties
    , let's plot the series, the
    series differencied 1 and 2
    times, and test stationarity
    with ADF test

# In[56]:


M['L_diff'] = M.Load.diff()


# In[18]:


plt.plot(M.L_diff)


# In[19]:


M['L_diff_2'] = M.L_diff.diff()


# In[20]:


plt.plot(M.L_diff_2,color = 'k')
plt.xlabel('Year')
plt.ylabel('Load')
plt.title('Monthly avarage of
    Polish Electric Load from 2008
    to 2017')


# In[64]:


M = M.fillna(0)


# In[65]:


import statsmodels.api as sm
from statsmodels.tsa.arima_model
    import ARIMA

fig = plt.figure(figsize=(12,8))
ax1 = fig.add_subplot(211)
fig = sm.graphics.tsa.plot_acf(M.
    Load, ax=ax1)
ax2 = fig.add_subplot(212)
fig = sm.graphics.tsa.plot_pacf(M.
    Load, ax=ax2)


# In[66]:


fig = plt.figure(figsize=(12,8))
ax1 = fig.add_subplot(211)
fig = sm.graphics.tsa.plot_acf(M.
    L_diff, ax=ax1)
ax2 = fig.add_subplot(212)
fig = sm.graphics.tsa.plot_pacf(M.
    L_diff, ax=ax2)


# In[70]:


```

```
fig = plt.figure(figsize=(12,8))
ax1 = fig.add_subplot(211)
fig = sm.graphics.tsa.plot_acf(M.L_diff_2, ax=ax1, lags =25)
ax2 = fig.add_subplot(212)
fig = sm.graphics.tsa.plot_pacf(M.L_diff_2, ax=ax2, lags = 25)


# In[75]:


fig = plt.figure(figsize=(12,8))
ax11 = fig.add_subplot(321)
plt.plot(M.Load,color = 'k')

plt.title('Monthly avarage')

ax12 = fig.add_subplot(322)
fig = sm.graphics.tsa.plot_acf(M.Load, ax=ax12, lags =25)
ax21 = fig.add_subplot(323)
plt.plot(M.L_diff,color = 'k')

plt.title('Monthly avarage 1st order differencing')
ax22 = fig.add_subplot(324)
fig = sm.graphics.tsa.plot_acf(M.L_diff, ax=ax22, lags =25)
ax21 = fig.add_subplot(325)
plt.plot(M.L_diff_2,color = 'k')

plt.title('Monthly avarage 2nd order differencing')
ax22 = fig.add_subplot(326)
fig = sm.graphics.tsa.plot_acf(M.L_diff_2, ax=ax22, lags =25)
fig.tight_layout(pad=3.0)


# In[69]:


result = adfuller(M.Load)
print('ADF Statistic: {}'.format(result[0]))
print('p-value: {}'.format(result[1]))
print('Critical Values:')
for key, value in result[4].items():
    print('\t{}: {}'.format(key, value))


# In[76]:


result = adfuller(M.L_diff)
print('ADF Statistic: {}'.format(result[0]))
print('p-value: {}'.format(result[1]))
print('Critical Values:')
for key, value in result[4].items():
    print('\t{}: {}'.format(key, value))


# In[77]:


result = adfuller(M.L_diff_2)
print('ADF Statistic: {}'.format(result[0]))
print('p-value: {}'.format(result[1]))
print('Critical Values:')
for key, value in result[4].items():
    print('\t{}: {}'.format(key, value))


# # ARMA

# First of all we try to build An ARMA model on the data

# In[24]:


N = len(M.Load)
split = 0.85
training_size = round(split*N)
test_size = round((1-split)*N)

```

```
series = M.Load[:training_size]


# In[25]:


def optimize_ARIMA(order_list, exog):
    """
        Return dataframe with parameters and corresponding AIC

        order_list - list with (p, d, q) tuples
        exog - the exogenous variable
    """

    results = []

    for order in tqdm_notebook(order_list):
        try:
            model = SARIMAX(exog, order=order).fit(disp=-1)
        except:
            continue

        aic = model.aic
        results.append([order, model.aic])

    result_df = pd.DataFrame(results)
    result_df.columns = ['(p, d, q)', 'AIC']
    #Sort in ascending order, lower AIC is better
    result_df = result_df.sort_values(by='AIC', ascending=True).reset_index(drop=True)

    return result_df


# In[26]:


ps = range(0, 10, 1)
d = 0
qs = range(0, 10, 1)

# Create a list with all possible combination of parameters
parameters = product(ps, qs)
parameters_list = list(parameters)

order_list = []

for each in parameters_list:
    each = list(each)
    each.insert(1, d)
    each = tuple(each)
    order_list.append(each)

result_df = optimize_ARIMA(order_list, exog = series)

result_df


# Try to fit the best model according to AIC

# In[76]:


best_model = SARIMAX(series, order=(6,2,6)).fit()
print(best_model.summary())
s_best_model = SARIMAX(series, order=(6,0,8)).fit()
print(s_best_model.summary())


# In[77]:


best_model.plot_diagnostics(figsize=(15,12))
s_best_model.plot_diagnostics(figsize=(15,12))


# In[29]:


```

```
original_1 = M.L_diff_2.cumsum() + M.L_diff.max()-M.L_diff_2.cumsum().max()
original_1[0]=M.L_diff[0]
original_2 = original_1.cumsum() + M.Load.max()- original_1.cumsum().max()
original_2[0] = M.Load[0]
plt.plot(M.Load)
plt.plot(original_2)


# Let's see how forecast performs

# In[86]:



fore_l= test_size -1
forecast = best_model.get_prediction(start=training_size, end=training_size+fore_l)
forec = forecast.predicted_mean
ci = forecast.conf_int(alpha=0.05)

s_forecast = s_best_model.get_prediction(start=training_size, end=training_size+fore_l)
s_forec = s_forecast.predicted_mean
s_ci = forecast.conf_int(alpha=0.05)

fig, ax = plt.subplots(figsize=(16,8), dpi=300)
x0 = M.Load.index[0:training_size]
x1=M.Load.index[training_size:training_size+fore_l+1]
#ax.fill_between(forec, ci['lower Load'], ci['upper Load'])
plt.plot(x0, M.Load[0:training_size],'k', label = 'Load')

plt.plot(M.Load[training_size:training_size+fore_l], '.k', label = 'Actual')

forec = pd.DataFrame(forec, columns=['f'], index = x1)
forec.f.plot(ax=ax,color = 'Darkorange',label = 'Forecast (d = 2)')
ax.fill_between(x1, ci['lower Load'], ci['upper Load'],alpha=0.2, label = 'Confidence inerval (95%)',color='grey')

s_forec = pd.DataFrame(s_forec, columns=['f'], index = x1)
s_forec.f.plot(ax=ax,color = 'green',label = 'Forecast  (d = 0)')
#ax.fill_between(x1, s_ci['lower Load'], s_ci['upper Load'], alpha=0.2, label = 'Confidence inerval (95%)',color='grey')


plt.legend(loc = 'upper left')
plt.show()


# In[89]:


from sklearn.metrics import mean_squared_error
RMSE_Arima = np.sqrt(mean_squared_error(M.Load[training_size:training_size+fore_l],forec.f[0:len(forec)-1]))
print('RMSE : {:.02f}'.format(RMSE_Arima))


# In[90]:


RMSE_Arima/M.Load.max()


# In[91]:


from sklearn.metrics import mean_squared_error
RMSE_Arima = np.sqrt(
```

```
mean_squared_error(M.Load[training_size:training_size+fore_l],s_forec.f[0:len(forec)-1]))
print('RMSE : {:.02f}'.format(RMSE_Arima))


# In[92]:


RMSE_Arima/M.Load.max()


# # SARIMA

# Let's see if taking care of seasonality could help building a better model, tre process is very similar to the one used for ARIMA modeling, however a decomposition tool helped us checking the stationarity components.

# In[100]:


def get_stationarity(timeseries, window):

    # rolling statistics
    rolling_mean = timeseries.rolling(window=window).mean()
    rolling_std = timeseries.rolling(window=window).std()

    # rolling statistics plot
    original = plt.plot(timeseries, color='blue', label='Original')
    mean = plt.plot(rolling_mean, color='red', label='Rolling Mean')
    std = plt.plot(rolling_std, color='black', label='Rolling Std')
    plt.legend(loc='best')
    plt.title('Rolling Mean & Standard Deviation')
    plt.show(block=False)

    # Dickey Fuller test:
    result = adfuller(timeseries)
    print('ADF Statistic: {}'.format(result[0]))
    print('p-value: {}'.format(result[1]))
    print('Critical Values:')
    for key, value in result[4].items():
        print('\t{}: {}'.format(key, value))


# In[95]:


from statsmodels.tsa.seasonal import seasonal_decompose

result = seasonal_decompose(M.Load, model='additive')
result.plot()
plt.show()


# In[101]:


len(S_series),len(RM_Load)


# In[102]:


N = len(M.Load)
split = 0.85
training_size = round(split*N)
test_size = round((1-split)*N)

S_series = M.Load[:training_size]


# In[103]:


len(RM_Load)
```

```
# In[105]:


window = 6
rolling_mean = S_series.rolling(window=window).mean()
RM_Load = S_series.diff(window)
RM_Load.dropna(inplace=True)
get_stationarity(RM_Load,window)
plt.plot(RM_Load)
plt.plot(S_series)


# In[106]:


validates = range(1,13,1)
p_0 = []
for v in validates:
    rolling_mean = S_series.rolling(window=v).mean()
    RM = S_series.diff(v)
    RM.dropna(inplace=True)
    result = adfuller(RM)
    p_0.append(result[1])


# In[111]:


p_value = np.full(len(validates), fill_value= 0.05)
plt.plot(validates,p_0,'.',label = 'P value')
plt.plot(validates,p_value,'--', label= 'Trashold (0.05)')
plt.xlabel('Seasonal difference')
plt.legend()
#plt.ylim(0,0.01)


# In[644]:


fig = plt.figure(figsize=(12,8))
ax1 = fig.add_subplot(211)
fig = sm.graphics.tsa.plot_acf(RM_Load, ax=ax1, lags =25)
ax2 = fig.add_subplot(212)
fig = sm.graphics.tsa.plot_pacf(RM_Load, ax=ax2, lags = 25)


# In[645]:


fig = plt.figure(figsize=(12,8))
ax3 = fig.add_subplot(311)
ax3.plot(RM_Load.diff().dropna().diff().dropna())
ax1 = fig.add_subplot(312)
fig = sm.graphics.tsa.plot_acf(RM_Load.diff().dropna().diff().dropna(), ax=ax1,lags =25)
ax2 = fig.add_subplot(313)
fig = sm.graphics.tsa.plot_pacf(RM_Load.diff().dropna().diff().dropna(), ax=ax2,lags=25)
fig.tight_layout(pad=3.0)


# In[121]:


def optimize_SARIMA(parameters_list, d, D, s, exog):
    """
        Return dataframe with parameters, corresponding AIC and SSE

        parameters_list - list with (p, q, P, Q) tuples
        d - integration order
        D - seasonal integration order
        s - length of season
        exog - the exogenous variable
    """

    results = []

    for param in tqdm_notebook(parameters_list):
        try:
```

```python
                model = SARIMAX(exog, order=(param[0],d, param[1]), seasonal_order=(param[2], D, param[3], s)).fit(disp=-1)
            except:
                continue

            aic = model.aic
            results.append([param, aic])

    result_df = pd.DataFrame(results)
    result_df.columns = ['(p,q)x(P,Q)', 'AIC']
    #Sort in ascending order, lower AIC is better
    result_df = result_df.sort_values(by='AIC', ascending=True).reset_index(drop=True)

    return result_df


# In[122]:


p = range(0, 4, 1)
d = 0
q = range(0, 4, 1)
P = range(0, 4, 1)
D = 1
Q = range(0, 4, 1)
s = 6
parameters = product(p, q, P, Q)
parameters_list = list(parameters)
print(len(parameters_list))


# In[123]:


result_df = optimize_SARIMA(parameters_list, d, D, s , S_series)
result_df


# Again building two models for comparison

# In[113]:


best_model = SARIMAX(S_series, order=(3, 2, 1), seasonal_order=(1, 1, 2, 6)).fit(disp=-1)
print(best_model.summary())
s_best_model = SARIMAX(S_series, order=(3,0, 1), seasonal_order=(1, 1, 2, 6)).fit(disp=-1)
print(s_best_model.summary())


# In[120]:


best_model.plot_diagnostics(figsize=(15,12));


# In[115]:


fore_l= test_size -1
forecast = best_model.get_prediction(start=training_size, end=training_size+fore_l)
forec = forecast.predicted_mean
ci = forecast.conf_int(alpha=0.05)

s_forecast = s_best_model.get_prediction(start=training_size, end=training_size+fore_l)
s_forec = s_forecast.predicted_mean
s_ci = forecast.conf_int(alpha=0.05)

fig, ax = plt.subplots(figsize=(16,8), dpi=300)
x0 = M.Load.index[0:training_size]
x1=M.Load.index[training_size:training_size+fore_l+1]
#ax.fill_between(forec, ci['lower Load'], ci['upper Load'])
```

```python
plt.plot(x0, M.Load[0:training_size],'k', label = 'Load')

plt.plot(M.Load[training_size:training_size+fore_l], '.k', label = 'Actual')

forec = pd.DataFrame(forec, columns=['f'], index = x1)
forec.f.plot(ax=ax,color = 'Darkorange',label = 'Forecast (d = 2)')
ax.fill_between(x1, ci['lower Load'], ci['upper Load'],alpha=0.2, label = 'Confidence inerval (95%)',color='grey')

s_forec = pd.DataFrame(s_forec, columns=['f'], index = x1)
s_forec.f.plot(ax=ax,color = 'green',label = 'Forecast  (d = 0)')
#ax.fill_between(x1, s_ci['lower Load'], s_ci['upper Load'], alpha=0.2, label = 'Confidence inerval (95%)',color='grey')


plt.legend(loc = 'upper left')
plt.show()



# In[116]:


RMSE_Sarima = np.sqrt(mean_squared_error(M.Load[training_size:training_size+fore_l],forec.f[0:len(forec)-1]))
print('RMSE : {:.02f}'.format(RMSE_Sarima))


# In[117]:


RMSE_Sarima/M.Load.max()


# In[124]:


RMSE_Sarima = np.sqrt(mean_squared_error(M.Load[training_size:training_size+fore_l],s_forec.f[0:len(forec)-1]))
print('RMSE : {:.02f}'.format(RMSE_Sarima))


# In[125]:


RMSE_Sarima/M.Load.max()
```

## 4.6 Deconvolution.ipynb

```python
#!/usr/bin/env python
# coding: utf-8

# # Deconvolution

# In[79]:


#Importing the libraries to watch the 'fits' image and get the data array
import astropy
#import plotly.graph_objects as go
from astropy.io import fits
#Importing a library that is useful to read the original file
import pandas as pd
import pylab as plb
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
from scipy import asarray as ar,exp
#Importing a visual library with some illustrative set up
import matplotlib.pyplot as plt
import matplotlib.colors as mcolors
from matplotlib import cm
import numpy as np
```

```
import math
import seaborn as sns
plt.style.use('fivethirtyeight')
plt.rcParams['font.family'] = 'sans-serif'
plt.rcParams['font.serif'] = 'Ubuntu'
plt.rcParams['font.monospace'] = 'Ubuntu Mono'
plt.rcParams['font.size'] = 14
plt.rcParams['axes.labelsize'] = 12
plt.rcParams['axes.labelweight'] = 'bold'
plt.rcParams['axes.titlesize'] = 12
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12
plt.rcParams['legend.fontsize'] = 12
plt.rcParams['figure.titlesize'] = 12
plt.rcParams['image.cmap'] = 'jet'
plt.rcParams['image.interpolation'] = 'none'
plt.rcParams['figure.figsize'] = (16, 8)
plt.rcParams['lines.linewidth'] = 2
plt.rcParams['lines.markersize'] = 8
plt.rcParams["axes.grid"] = False

colors = ['xkcd:pale orange', 'xkcd:sea blue', 'xkcd:pale red', 'xkcd:sage green', 'xkcd:terra cotta', 'xkcd:dull purple', 'xkcd:teal', 'xkcd: goldenrod', 'xkcd:cadet blue',
'xkcd:scarlet']
cmap_big = cm.get_cmap('Spectral', 512)
cmap = mcolors.ListedColormap(cmap_big(np.linspace(0.7, 0.95, 256)))
bbox_props = dict(boxstyle="round,pad=0.3", fc=colors[0], alpha=.5)


# First step we import filtered dataset. (wavelet)

# In[80]:


dataset = pd.read_csv('reconstruction.csv')
dataset = dataset.drop(columns=['Unnamed: 0'])


# In[81]:


data=pd.read_csv('deconv.csv',sep=';')


# In[82]:


data=data.rename(columns={'Data':'Day','Godzina':'hour','Minuty':'minute','Wolumen':'Load'})


# In[83]:


data['Load'] = dataset.R


# In[84]:


data.head()


# Coverting data into DateTime objects.

# In[85]:


data.Day = pd.to_datetime(data.Day)


# In[86]:
```

```
data.Day = pd.to_datetime(data.Day)
data['year'] = data.Day.dt.year
data['month'] = data.Day.dt.month
data['day'] = data.Day.dt.day


# In[87]:


data['month'] = data.Day.dt.month


# In[88]:


data['day'] = data.Day.dt.day


# In[89]:


data['weekday']=data.Day.dt.day_name()


# In[91]:


data.head()


# Distinguish Holidays from weekdays

# In[92]:


import holidays
pl_holidays = holidays.Poland()


# In[93]:


from datetime import date
Days = data.day.tolist()
Months = data.month.tolist()
Years = data.year.tolist()
hd = []
for i in range(len(data)):
    if date(Years[i], Months[i], Days[i]) in pl_holidays :
            hd.append(1)
    else :
        hd.append(0)
data['holidays'] = hd


# In[94]:


data.head()


# In[95]:


weekdays = data.weekday.drop_duplicates().tolist()
weekdays = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']


# Building kernel functions for weekdays and weekend without holidays.

# In[97]:


Mean = []
for w in weekdays:
    W = data[data['weekday'] == w]
    W = W[W['holidays'] == 0]
    W_Day = W.Day.drop_duplicates().tolist()
    mean = pd.DataFrame()
    mean = np.zeros(96)
    k = 0
    for d in W_Day:
        value = W[W['Day'] == d].Load
        try :
            mean =  mean + np.asarray(value)
```

```
            k = k+1
        except :
            continue

    mean = mean/k
    Mean.append(mean)


# In[98]:


plt.figure(1)
plt.subplot(711)
plt.plot(Mean[0])
plt.subplot(712)
plt.plot(Mean[1])
plt.subplot(713)
plt.plot(Mean[2])
plt.subplot(714)
plt.plot(Mean[3])
plt.subplot(715)
plt.plot(Mean[4])
plt.subplot(716)
plt.plot(Mean[5])
plt.subplot(717)
plt.plot(Mean[6])


# In[99]:


d=data.Day.drop_duplicates()


# In[101]:


Ferial = np.array(Mean[0]) + np.array(Mean[1]) + np.array(Mean[2]) + np.array(Mean[3]) + np.array(Mean[4])
Ferial=Ferial/5.
Ferial=Ferial/Ferial.sum()


# In[102]:


plt.plot(Ferial),Ferial.mean()
```

```


# In[103]:


we= np.array(Mean[5])+np.array(Mean[6])
we=we/2.
we=we/we.sum()


# In[104]:


day=data.Day.drop_duplicates()


# ## Building the delta signal for convolution

# Some days are longer than others, while other days were snaller.
# We fix this in the following way.

# In[105]:


MAX=[]
PROB_D=[]
DAY=np.array(day)
for t in range(len(DAY)-1):
    startdata=data[(data.Day==DAY[t])].index.tolist()[0]
    enddata=data[(data.Day==DAY[t+1])].index.tolist()[0]
    if enddata-startdata!=96:
        PROB_D.append(DAY[t])

    arr_= np.array(data.Load.loc[startdata:enddata])

    #arr_= arr_-arr_.mean()
    MAX.append(arr_.max())
    #print(t,len(DAY)-1)


# In[106]:


```

```
OLD_PROB_D=PROB_D.copy()


# In[107]:


fix_data=data.copy()
I=np.arange(0,len(OLD_PROB_D)+1,1)
for i in range(len(I)-1):
    if i%2==0:
        to_add=fix_data.loc[fix_data[(fix_data.Day==OLD_PROB_D[I[i+1]]) & (fix_data.hour==2)].index.tolist()[0:4]].Load
        line = fix_data.loc[to_add.index[0]:to_add.index[0]+3]
        start=fix_data[(fix_data.Day==OLD_PROB_D[I[i]]) & (fix_data.hour==1)].index.max()+1
        try_data=fix_data.copy()
        try_data=try_data.drop(np.arange(start,len(data)))
        fix_data=try_data.append(line,ignore_index=True)
        fix_data.loc[len(fix_data)-4:len(fix_data)].Day=np.repeat(OLD_PROB_D[I[i]],4)
        fix_data=fix_data.append(data.loc[fix_data.index.max()-3:len(data)])
        fix_data=fix_data.drop(line.index)


# Fix data is used to build Delta Signal

# In[108]:


data=fix_data


# In[109]:


MAX=[]
PROB_D=[]
#data=data.reset_index()
RANGE=np.arange(0,len(fix_data),96)
for t in RANGE:
    MAX.append(fix_data.loc[t:t+96-1].Load.max())


# In[110]:


def build_day(k):
    day=np.zeros(96)
    day[0]=k
    return day


# In[111]:


DELTA=[]
for m in MAX:
    DELTA.append(build_day(m))
#DELTA=np.abs(DELTA)


# In[112]:


DELTA=np.array(DELTA).ravel()


# In[118]:


data['r']=DELTA


# In[119]:


fix_data['r']=DELTA


# In[143]:


plt.plot(np.array(fix_data.r[96:96*30]),'Darkorange', label
```

```
    = 'Delta signal')
plt.plot(np.array(fix_data.Load[96:96*30]),'k', label = 'Time Series')
plt.grid(True)
plt.ylabel('Load')
plt.xlabel('Time')
plt.xticks(np.arange(96,96*30,96),np.arange(0,30))
plt.legend()
plt.show()


# In[121]:


data=fix_data


# In[122]:


YEARS=data.year.drop_duplicates().tolist()


# Each specific day of the months for each year has been stored
# e.g.
# The first Monday of the third week of May
# the second Friday of the first week of April

# In[123]:


TOT=[]
for m in range(1,13):
    DAYS=[]
    for d in weekdays:
        mon=data[data['weekday']==d]
        W=[]
        J=np.arange(0,96*7,96)
        K=[]
        for j in range(len(J)):
            j_rel=np.zeros(96)
            k=0
            for y in range(len(YEARS)-1):
                try:
                    add=np.array(mon[(mon['month']==m)&(mon.year==YEARS[y])].reset_index().loc[J[j]:J[j]+95].r)
                    j_rel=j_rel+add
                    k=k+1
                except:
                    continue
            K.append(k)
            W.append(j_rel/k)
        DAYS.append(W)

    TOT.append(DAYS)


# The calendar of 2016 has been buit

# In[144]:



cal=pd.DataFrame({'Day':data['Day'],'month':data['month'],'year':data['year'],'Weekday':data['weekday']})
cal['hour']=data.hour
cal['minute']=data.minute
cal=cal[cal['year']==2016]


# In[145]:



NEW_Z=[]
for k in range(1,13):
    mese=cal[cal['month']==k]
    mese=mese.reset_index().drop(columns=['index'])
    z=[]
    for j in range(1,6):
        sett=mese.loc[(j-1)*7*96:96*(j)*7]
        #print(j-1)
        ZERO=np.zeros(len(sett)-1)+
```

```
   j-1
        z.append(ZERO)
    new_z=[]
    for i in z:
        for j in i:
            new_z.append(j)
    new_z=new_z[0:len(mese)]+[new_z[len(new_z)-1]]
    NEW_Z.append(new_z)



# In[146]:



ZERO_FIN=[]
for i in NEW_Z:
    for j in i:
        ZERO_FIN.append(j)


# In[147]:


cal['n']=ZERO_FIN


# In[148]:


tot_cal=cal.drop(columns=['hour','minute'])
tot_cal=tot_cal.drop_duplicates(subset='Day')


# In[149]:


tot_cal=tot_cal.reset_index()


# In[150]:


def week_to_number(day):
    if day=='Monday':
        i=0
    if day=='Tuesday':
        i=1
    if day=='Wednesday':
        i=2
    if day=='Thursday':
        i=3
    if day=='Friday':
        i=4
    if day=='Saturday':
        i=5
    if day=='Sunday':
        i=6
    return i


# In[151]:


d_n=[]
for day in tot_cal.Weekday.tolist():
    d_n.append(week_to_number(day))


# In[152]:


tot_cal['d_n']=d_n


# The model of each day has been used to create a synthetic 2016

# In[153]:


SIGNAL=np.array([])
for m in range(1,13):
    mon=tot_cal[tot_cal['month']==m]
    mon=mon.reset_index()

    for i in range(len(mon)):
            dn=int(mon.loc[i]['d_n'])
            N=int(mon.loc[i]['n'])
            SIGNAL=np.append(SIGNAL,TOT[m-1][dn][N])

```

```python

# Debugging the split years

# In[154]:


bis_1=data[data.year==2008]
bis_2=data[data.year==2012]
first=np.array(bis_1[bis_1['Day']=='2008-02-29'].r)
second=np.array(bis_2[bis_2['Day']=='2012-02-29'].r)
bis=(first+second)/2.


# In[155]:


a=np.argwhere(np.isnan(SIGNAL)).ravel().min()
b=np.argwhere(np.isnan(SIGNAL)).ravel().max()+1
SIGNAL[a:b]=bis


# In[157]:


OLD_SIGNAL=SIGNAL.copy()


# In[158]:


SIGNAL=np.abs(OLD_SIGNAL)


# In[159]:


Ferial = np.array(Mean[0]) + np.array(Mean[1]) + np.array(Mean[2]) + np.array(Mean[3]) + np.array(Mean[4])
Ferial=Ferial/5.
Ferial=Ferial/Ferial.sum()
Old_Ferial=Ferial.copy()


# In[160]:


Ferial = np.array(Mean[0]) + np.array(Mean[1]) + np.array(Mean[2]) + np.array(Mean[3]) + np.array(Mean[4])
Ferial=Ferial/5.
Ferial=Ferial/Ferial.sum()
Ferial=Ferial+Ferial.mean()-Ferial.min()-Ferial.max()


# In[161]:


Ferial=Ferial/5.


# In[162]:


Ferial=Ferial/Ferial.sum()


# In[163]:


Ferial=Ferial+Ferial.mean()-Ferial.min()-Ferial.max()


# Performing the convolution with Ferial days

# In[166]:


X=signal.convolve(SIGNAL,Ferial)
X=X*(data.Load.max())/X.max()


# In[167]:


test=np.array(data[data.year==2016].Load)


```

```
# In[168]:


X=X[0:len(X)-95]


# In[169]:


plt.plot(X[60*96:90*96],color='blue')
plt.plot(np.array(test)[60*96:90*96],color='red')
#error=np.array(test)[12*96:50*96]-original
#plt.plot(np.abs(error),color='black')


# Building convolution for weekend days

# In[170]:


start=20*96
end=(20+7*40)*96
prev=X[start:end]
t=np.arange(0,len(prev),1)


# In[173]:


sat= np.array(Mean[6])
sat=sat-sat.min()
sat=sat/sat.sum()
sat=sat-sat.mean()


# In[204]:


sun= np.array(Mean[6])
sun=sun-sun.min()
sun=sun/sun.sum()
sun=sun-sun.mean()
sat= np.array(Mean[5])
sat=sat-sat.min()
sat=sat/sat.sum()
sat=sat-sat.mean()
plt.plot(Ferial,'k',label = 'Ferial')
plt.plot(sun,'red',label = 'Sunday')
plt.plot(sat,'Darkorange',label = 'Saturday')
plt.legend()
plt.grid(True)
plt.ylabel('Kernel Function')
plt.xlabel('Time [Hours]')
plt.xticks(np.arange(0,96,4),np.arange(0,24))
plt.show()


# Prediction using weekend days kernels for entire calendar.

# In[175]:


SUN=signal.convolve(sun,SIGNAL)
SUN=SUN*data[data.year!=2016].Load.max()/SUN.max()
SAT=signal.convolve(sat,SIGNAL)
SAT=SAT*data[data.year!=2016].Load.max()/SAT.max()


# In[176]:


plt.plot(SAT[60*96:90*96])
plt.plot(test[60*96:90*96])


# Dataset of the 3 Kernels reconstruction

# In[ ]:


recons_data=data[data['year']==2016].copy()
recons_data['PSFFerial']=X[0:len(recons_data)]
recons_data['PSFSunday']=SUN[0:len(
```

```
    recons_data)]
recons_data['PSFSaturday']=SAT[0:len(recons_data)]


# In[178]:


recons_data


# In[180]:


psf_data=pd.DataFrame()
psf_data['weekday']=recons_data['weekday']
psf_data['Ferial']=recons_data.PSFFerial
psf_data['Sunday']=recons_data.PSFSunday
psf_data['Saturday']=recons_data.PSFSaturday
psf_data=psf_data.reset_index()


# In[182]:


final_recons=np.array(psf_data.Ferial)
for i in range(35136):
     day_data=str(psf_data.loc[i].weekday)
     if day_data=='Sunday':
         final_recons[i]=psf_data.loc[i].Sunday
     if day_data=='Saturday':
         final_recons[i]=psf_data.loc[i].Saturday



# Final Prediction

# In[226]:


plt.plot(final_recons[60*96:90*96],color='Darkorange',label='Model')
plt.plot(test[60*96:90*96],color='k',label = 'Data')
plt.grid(True)
plt.legend()
plt.xticks(np.arange(0*96,31*96,96),np.arange(0,31))
plt.xlabel('Days [March 2016]')
plt.ylabel('Load')


# In[214]:


from sklearn.metrics import mean_squared_error


# In[216]:


finaldata=data[data['year']==2016].copy()
#finaldata['Prediction']=final_recons


# In[217]:


finaldata.to_csv('pred.csv')


# In[218]:


finaldata=pd.read_csv('pred.csv')


# In[1158]:


test=np.array(finaldata.Load)
pred=np.array(final_recons)


# In[29]:

```

```python

from sklearn.metrics import
    mean_squared_error as mse


# In[41]:


PERF=[np.corrcoef(test-pred,test)
    [0][1], np.sqrt(mse(pred
    [0*96:30*96],test[0*96:30*96]))
    ,np.sqrt(mse(pred[30*96::],test
    [30*96::]))/test[30*96::].max()
    ]


# In[45]:


print ('The correlation between the
     error and the original signal
    is '+str(PERF[0]))
print ('The RMSE for the first
    month of data is ' + str(PERF
    [1]))
print ('This RMSE is the '+str(PERF
    [2]*100) +'% of the maximum')
```

# Bibliography


[1] Del Moro Dario, De Gasperis Giancarlo. *Lezione 2 - The Nyquist-Shannon sampling theorem*. 2020. URL: http://people.fisica.uniroma2.it/~solare/en/wp-content/uploads/2018/11/Lez_2.pdf.

[2] Del Moro Dario, De Gasperis Giancarlo. *Lezione 7- Wavelets*. 2020. URL: http://people.fisica.uniroma2.it/~solare/en/wp-content/uploads/2018/11/Lez_7_Wavelets.pdf.

[3] Piero Paialunga. "Hey wavelet, turn that noise down!" In: *Towards Data Science* (2020). URL: https://towardsdatascience.com/hey-wavelet-turn-that-noise-down-80a30796dfcf.

[4] Piero Paialunga. "If history repeats itself, Fourier Transform is a key". In: *Towards Data Science* (2020). URL: https://towardsdatascience.com/if-history-repeats-itself-fourier-transform-is-a-key-a593ddfa246e.

[5] Wikipedia contributors. *Chi-squared test — Wikipedia, The Free Encyclopedia*. [Online; accessed 4-November-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=Chi-squared_test&oldid=983024096.

[6] Wikipedia contributors. *Hann function — Wikipedia, The Free Encyclopedia*. [Online; accessed 30-October-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=Hann_function&oldid=970339234.

[7] Wikipedia contributors. *P-value — Wikipedia, The Free Encyclopedia*. [Online; accessed 4-November-2020]. 2020. URL: https://en.wikipedia.org/w/index.php?title=P-value&oldid=979081903.