**Time series Preprocessing**

[[Fernando Luna](https://medium.com/@lu.fernando2901?source=post_page-----946ccaff3ff1--------------------------------)](https://medium.com/@lu.fernando2901?source=post_page-----946ccaff3ff1--------------------------------)

[Fernando Luna](https://medium.com/@lu.fernando2901?source=post_page-----946ccaff3ff1--------------------------------)

·

Follow

32 min read

·

Dec 1

16

authors:

[Eliud Rodríguez](https://medium.com/u/760bdc41c403?source=post_page-----946ccaff3ff1--------------------------------)

,

[Fernando Luna](https://medium.com/u/ff4119735686?source=post_page-----946ccaff3ff1--------------------------------)

,

[Osiris A. Izaguirre](https://medium.com/u/2c3e88611930?source=post_page-----946ccaff3ff1--------------------------------)

**Handling Missing Data and Outliers.**

**Initial Data.**

To begin, I used a dataset containing measurements of different parameters related to air quality. A fundamental step in data analysis is understanding the integrity of the data we have. To do this, I started by counting missing values in our dataset:

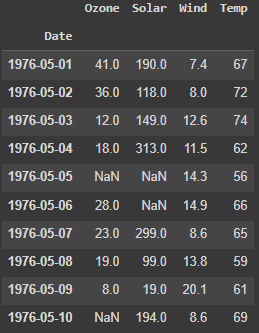
The code below provides us with a quick overview of how many missing values exist in each column of our data. Identifying these missing values is crucial as they can significantly impact our analyses and results.

# Contar valores faltantes  
missing\_values\_count = df.isna().sum()  
print(missing\_values\_count)



In the code below, we’re showing the initial rows of our air quality dataset. This gives us a glimpse of the data’s structure and allows us to understand its format and content.

# Display the first few rows of the dataset to get an overview  
df.head(10)



The df.shape command provides us with the dimensions of our dataset. It tells us how many rows and columns our data contains. Understanding the dataset's size is essential as it gives us an idea of the data's volume and structure, which is crucial for effective analysis and visualization.

# Shape of data  
df.shape



In the code below, we’re determining the frequency of our time series data. Knowing the data frequency is crucial when working with time-series datasets. It tells us how frequently data points are recorded, which is essential for various time-based analyses, such as trend detection and periodicity assessment. Understanding the data frequency helps us interpret and process the data accurately.

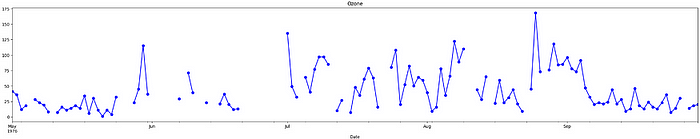
# Calculate the time difference between consecutive data points  
time\_diff = df.index.to\_series().diff()  
  
# Find the most common time difference  
most\_common\_freq = time\_diff.mode().iloc[0]  
  
# Display the frequency of the data  
print(f"Data Frequency: {most\_common\_freq}")



**Data Visualization.**

Next, I visualized one of the key parameters in our dataset, “Ozone,” to better understand its distribution and detect outliers:

# Visualizar el parámetro "Ozono"  
df['Ozone'].plot(title='Ozono', marker='o', color='blue', figsize=(30, 5))  
plt.show()



This visualization helps us identify patterns in the data and potential outliers that may require special treatment.

**Data Imputation.**

After analyzing missing data and visualizing values, the next step is to address missing values and outliers. Data imputation is a crucial process where missing values are filled in with estimates based on existing data. There are several techniques to do this, such as mean or median imputation or even using machine learning models. I explored different imputation approaches to show in this post.

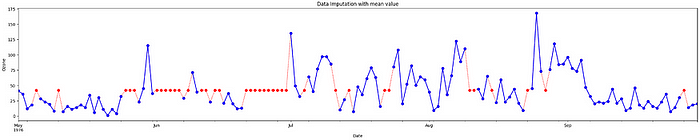
In the upcoming sections of this article, I will delve into specific data imputation techniques and outlier handling methods I explored and how to apply them to our air quality dataset. Stay tuned for more details on addressing these common challenges in data analysis!

**Data Imputation with Mean.**

In our continuous exploration of data imputation techniques to tackle missing values within our air quality dataset, we turn our attention to a fundamental yet powerful method: imputing missing values with the mean. This approach is a cornerstone of data preprocessing and provides a practical solution for handling data gaps.

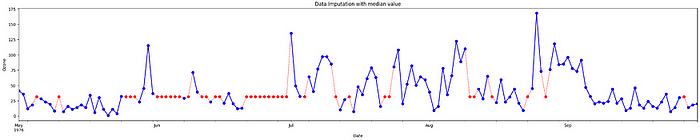
Here’s how I applied this method to our dataset:

# Impute missing values with the mean  
  
# Create a copy of the DataFrame  
mean\_imputed = df.copy(deep=True)  
  
# Calculate the mean of the 'Ozone' column  
mean\_value = mean\_imputed['Ozone'].mean()  
  
# Fill missing values with the mean  
mean\_imputed['Ozone'].fillna(mean\_value, inplace=True)  
  
# Plot the imputed DataFrame in red dotted style  
mean\_imputed['Ozone'].plot(color='red', marker='o', linestyle='dotted', figsize=(30, 5))  
  
# Plot the original air quality DataFrame with title  
df['Ozone'].plot(title='Data Imputation with mean value', ylabel='Ozone', marker='o', color='blue', figsize=(30, 5))



In our exploration of data imputation techniques for addressing missing values in our air quality dataset, we introduce a robust method: imputing missing values with the median. This approach offers a resilient solution for handling missing data, particularly in scenarios where data robustness is crucial. Median imputation is preferred when dealing with outlier-prone data, non-normal data distributions, or when preserving data distribution is paramount. It provides a stable estimate of central tendency while minimizing the influence of extreme values.

# Impute missing values with the median  
  
# Create a copy of the DataFrame  
median\_imputed = df.copy(deep=True)  
  
# Calculate the median of the 'Ozone' column  
median\_value = median\_imputed['Ozone'].median()  
  
# Fill missing values with the median  
median\_imputed['Ozone'].fillna(median\_value, inplace=True)  
  
# Plot the imputed DataFrame in red dotted style  
median\_imputed['Ozone'].plot(color='red', marker='o', linestyle='dotted', figsize=(30, 5))  
  
# Plot the original air quality DataFrame with title  
df['Ozone'].plot(title='Data Imputation with median value', ylabel='Ozone', marker='o', color='blue', figsize=(30, 5))

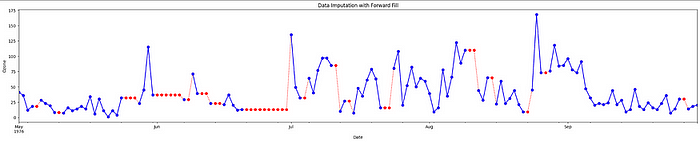


**Data Imputation with Forward Fill.**

One of the techniques I explored for handling missing data in our air quality dataset is called “forward fill.” This method is particularly useful when dealing with time series data or data where missing values can be reasonably estimated by the most recent available value.

Here’s how I applied the forward fill method to our dataset:

# Forward Fill Imputation  
  
# Create a copy of the DataFrame  
ffill\_imputed = df.copy(deep=True)  
  
# Impute missing values using forward fill method  
ffill\_imputed.fillna(method='ffill', inplace=True)  
  
# Plot the imputed DataFrame in red dotted style  
ffill\_imputed['Ozone'].plot(color='red', marker='o', linestyle='dotted', figsize=(30, 5))  
  
# Plot the original air quality DataFrame with title  
df['Ozone'].plot(title='Ozone', marker='o', color='blue', figsize=(30, 5))  
plt.show()

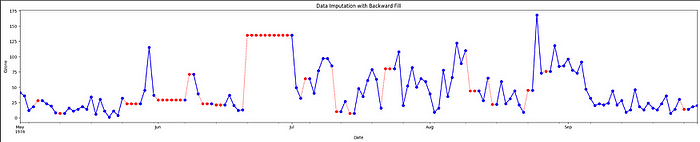


**Data Imputation with Backward Fill.**

In our quest to handle missing data effectively in the air quality dataset, we’ve explored another valuable technique called “backward fill” or “back fill.” This method is particularly useful when we want to fill missing values with the next available value in a column, which can be beneficial in scenarios where the data exhibits a certain pattern of continuity.

Here’s how I applied the backward fill method to our dataset:

# Backward Fill Imputation  
  
# Create a copy of the DataFrame  
bfill\_imputed = df.copy(deep=True)  
  
# Impute missing values using backward fill method  
bfill\_imputed.fillna(method='bfill', inplace=True)  
  
# Plot the imputed DataFrame in red dotted style  
bfill\_imputed['Ozone'].plot(color='red', marker='o', linestyle='dotted', figsize=(30, 5))  
  
# Plot the original air quality DataFrame with title  
df['Ozone'].plot(title='Ozone', marker='o', color='blue', figsize=(30, 5))  
plt.show()

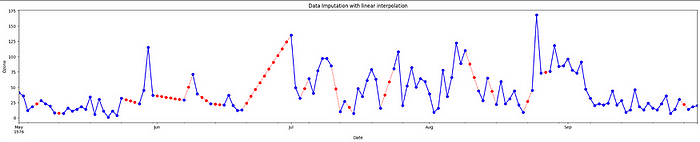


**Data Imputation with Linear Interpolation.**

In our journey to handle missing data in our air quality dataset, we’ve now explored yet another powerful technique: “linear interpolation.” Linear interpolation is a method that estimates missing values by creating a linear relationship between adjacent data points. This approach is particularly useful when there is a gradual change in the data over time.

Here’s how I applied linear interpolation to our dataset:

# Linear Interpolation Imputation  
  
# Create a copy of the DataFrame  
linear\_imputed = df.copy(deep=True)  
  
# Impute missing values using linear interpolation  
linear\_imputed.interpolate(method='linear', inplace=True)  
  
# Plot the imputed DataFrame in red dotted style  
linear\_imputed['Ozone'].plot(color='red', marker='o', linestyle='dotted', figsize=(30, 5))  
  
# Plot the original air quality DataFrame with title  
df['Ozone'].plot(title='Ozone', marker='o', color='blue', figsize=(30, 5))

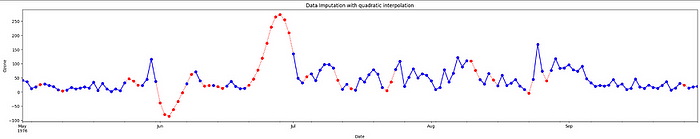


**Data Imputation with Quadratic Interpolation.**

In our ongoing exploration of methods for handling missing data in our air quality dataset, we’ve now delved into “quadratic interpolation.” Quadratic interpolation is a technique that estimates missing values by fitting a quadratic function to the neighboring data points. This approach is valuable when the data exhibits nonlinear patterns.

Here’s how I applied quadratic interpolation to our dataset:

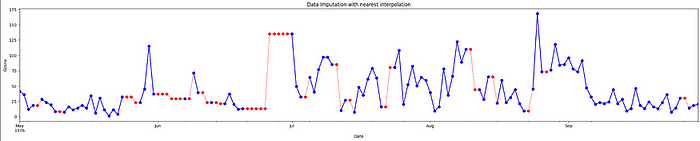
# Quadratic Interpolation Imputation  
  
# Create a copy of the DataFrame  
quadratic\_imputed = df.copy(deep=True)  
  
# Impute missing values using quadratic interpolation  
quadratic\_imputed.interpolate(method='quadratic', inplace=True)  
  
# Plot the imputed DataFrame in red dotted style  
quadratic\_imputed['Ozone'].plot(color='red', marker='o', linestyle='dotted', figsize=(30, 5))  
  
# Plot the original air quality DataFrame with title  
df['Ozone'].plot(title='Ozone', marker='o', color='blue', figsize=(30, 5))



**Data Imputation with Nearest Interpolation.**

In our ongoing exploration of data imputation techniques for handling missing data in our air quality dataset, we’ve ventured into “nearest interpolation.” Nearest interpolation is a method that estimates missing values by taking the value of the nearest neighboring data point. This approach can be valuable when the data exhibits abrupt changes or irregular intervals.

# Nearest Interpolation Imputation  
  
# Create a copy of the DataFrame  
nearest\_imputed = df.copy(deep=True)  
  
# Impute missing values using nearest interpolation  
nearest\_imputed.interpolate(method='nearest', inplace=True)  
  
# Plot the imputed DataFrame in red dotted style  
nearest\_imputed['Ozone'].plot(color='red', marker='o', linestyle='dotted', figsize=(30, 5))  
  
# Plot the original air quality DataFrame with title  
df['Ozone'].plot(title='Ozone', marker='o', color='blue', figsize=(30, 5))



**Data Imputation with K-Nearest Neighbors (KNN).**

In our ongoing exploration of data imputation techniques for handling missing data in our air quality dataset, we’ve now ventured into the world of K-Nearest Neighbors (KNN) imputation.

K-Nearest Neighbors (KNN) is a versatile algorithm used for both classification and regression tasks. However, it can also be employed for imputing missing values, which is particularly useful when dealing with incomplete datasets.

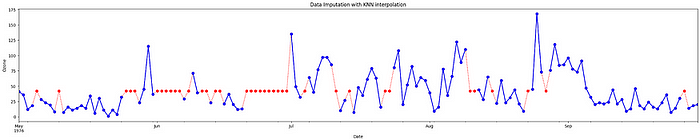
The core idea behind KNN imputation is straightforward:

**Locate Nearest Neighbors:** For each missing data point, identify its k-nearest neighbors in the dataset based on some similarity metric. In our case, we’ll focus on numerical features and use Euclidean distance to measure similarity.

**Impute Missing Values:** Calculate the weighted average (or another aggregation) of the known values from the identified neighbors. This calculated value is then used to replace the missing data point.

**Repeat for All Missing Values:** Iterate through all missing values and apply the same process to impute them.

# KNN Imputation  
from sklearn.impute import KNNImputer  
  
# Create a copy of the DataFrame for imputation  
knn\_imput = df.copy(deep=True)  
  
# Column to impute  
column\_to\_impute = 'Ozone'  
  
# Initialize the KNNImputer with the desired number of neighbors  
knn\_imputer = KNNImputer(n\_neighbors=5) # You can adjust the number of neighbors  
  
# Perform KNN imputation for the specified column  
knn\_imput[[column\_to\_impute]] = knn\_imputer.fit\_transform(knn\_imput[[column\_to\_impute]])  
  
# Plot the original 'Ozone' column  
df[column\_to\_impute].plot(title='Original ' + column\_to\_impute, marker='o')  
  
# Plot the imputed 'Ozone' column in red dotted style  
knn\_imput[column\_to\_impute].plot(title='Imputed ' + column\_to\_impute, marker='o', color='red', linestyle='dotted', figsize=(30, 5))  
  
# Plot the original air quality DataFrame with title  
df['Ozone'].plot(title='Data Imputation with nearest interpolation', ylabel='Ozone', marker='o', color='blue', figsize=(30, 5))

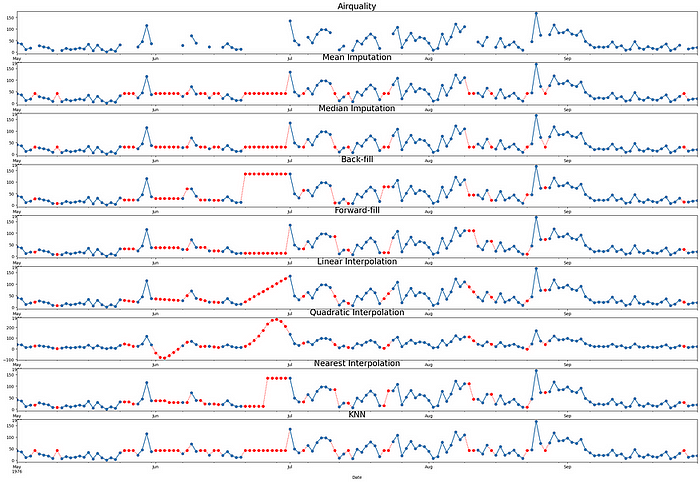


**Comparison of Imputation Methods.**

Throughout this article, we’ve explored several imputation methods for handling missing data in our air quality dataset. Now, let’s bring them all together in a comprehensive comparison to understand their respective impacts on the “Ozone” parameter.

Below, you’ll find a plot that illustrates how each imputation method affects the “Ozone” parameter compared to the original data. We’ve considered the following imputation techniques:

# Set nrows to 9 and ncols to 1 to accommodate the two new methods  
fig, axes = plt.subplots(9, 1, figsize=(30, 20))  
  
# Create a dictionary of interpolations including Mean and Median methods  
interpolations = {'Airquality': df,  
 'Mean Imputation': mean\_imputed,  
 'Median Imputation': median\_imputed,  
 'Back-fill': bfill\_imputed,  
 'Forward-fill': ffill\_imputed,  
 'Linear Interpolation': linear\_imput,  
 'Quadratic Interpolation': quadratic\_imput,  
 'Nearest Interpolation': nearest\_imput,  
 'KNN': knn\_imput  
 }   
# Loop over axes and interpolations  
for ax, df\_key in zip(axes, interpolations):  
 # Select and also set the title for a DataFrame  
 interpolations[df\_key].Ozone.plot(color='red', marker='o',  
 linestyle='dotted', ax=ax)  
 df.Ozone.plot(title=df\_key + ' - Ozone', marker='o', ax=ax)  
 # Set the title and increase the font size  
 ax.set\_title(df\_key, fontsize=20)  
plt.show()



**Handling Outliers.**

Outliers can greatly influence the results of our analysis. In my task, I researched how to effectively detect and handle these outliers. This can include techniques like outlier removal, data transformation, or the use of robust methods to mitigate their impact on our conclusions.

**Outlier Detection with IQR**

Outliers identified using this method are those that fall outside the range defined by the lower and upper bounds calculated from the IQR. The idea behind this approach is that outliers lie beyond certain multiples of the IQR away from the quartiles, suggesting that they are unusual or unexpected values in the context of the dataset.

**Calculate Quartiles**: First, we calculate the first quartile (Q1) and the third quartile (Q3) of the dataset. These quartiles divide the data into four equal parts, with Q1 representing the 25th percentile and Q3 representing the 75th percentile of the data.

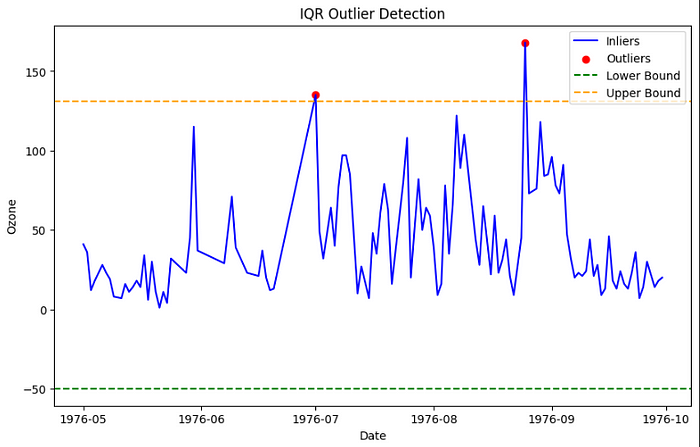
**Determine IQR**: The Interquartile Range (IQR) is then calculated as the difference between Q3 and Q1 (IQR = Q3 — Q1). The IQR essentially represents the spread or variability of the middle 50% of the data.

**Define Bounds**: Next, we establish lower and upper bounds to identify potential outliers. These bounds are calculated as follows:

Lower Bound = Q1–1.5 \* IQR  
Upper Bound = Q3 + 1.5 \* IQR

**Identify Outliers**: Any data points falling below the lower bound or above the upper bound are considered potential outliers. These are values that are significantly different from the central bulk of the data and are flagged as potential anomalies.

# Assuming 'Ozone' is the column of interest  
X = linear\_imput['Ozone'].values  
  
# Compute the IQR-based bounds  
Q1 = df['Ozone'].quantile(0.25)  
Q3 = df['Ozone'].quantile(0.75)  
IQR = Q3 - Q1  
lower\_bound = Q1 - 1.5 \* IQR  
upper\_bound = Q3 + 1.5 \* IQR  
  
# Identify outlier indices  
outlier\_indices = df[(df['Ozone'] < lower\_bound) | (df['Ozone'] > upper\_bound)].index  
  
# Plot the data with outliers marked in red  
plt.figure(figsize=(10, 6))  
  
# Plot inliers as a line plot  
plt.plot(df.index, X, color='blue', label='Inliers')  
  
# Scatter plot for outliers in red  
plt.scatter(outlier\_indices, df.loc[outlier\_indices, 'Ozone'], color='red', label='Outliers')  
  
# Plot the upper and lower bounds as horizontal lines  
plt.axhline(y=lower\_bound, color='green', linestyle='--', label='Lower Bound')  
plt.axhline(y=upper\_bound, color='orange', linestyle='--', label='Upper Bound')  
plt.xlabel('Date')  
plt.ylabel('Ozone')  
plt.title('IQR Outlier Detection')  
plt.legend()  
plt.show()



**Outlier Detection with Isolation Forest.**

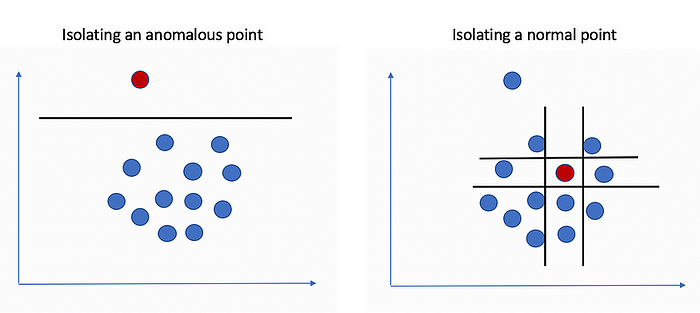
The Isolation Forest method is an anomaly detection technique that excels in identifying outliers by isolating them from the majority of data points. Unlike traditional methods that focus on the properties of inliers, Isolation Forest takes a novel approach: it assumes that outliers are rare and, therefore, easier to isolate.

Isolation Forest creates partitions (lines or splits) in a dataset by selecting random features and splitting values.

Data points that require fewer partitions (lines) to be isolated are considered anomalies or outliers.

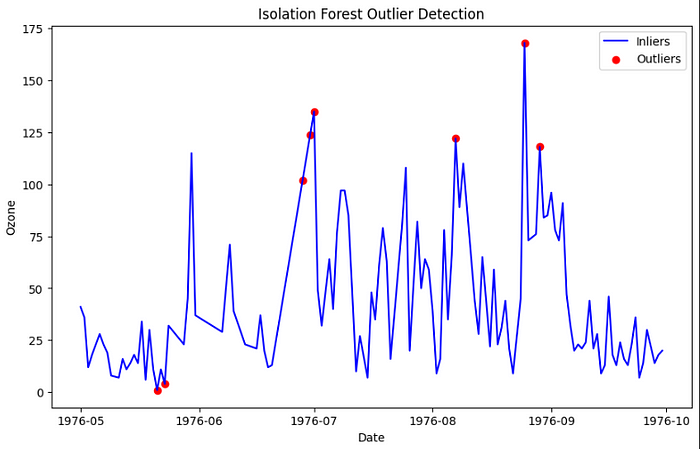
Conversely, data points that require more partitions (lines) to be isolated are considered inliers or normal data points.

So, in essence, Isolation Forest identifies outliers by quantifying how easily or quickly data points can be isolated from the majority of the data. Outliers are the data points that are isolated with minimal effort, while inliers are those that require more splits to be separated. This unique approach is what makes the Isolation Forest algorithm efficient and effective for anomaly detection tasks.



[Visual representation of anomalous and normal points isolations.](https://towardsdatascience.com/how-to-perform-anomaly-detection-with-the-isolation-forest-algorithm-e8c8372520bc)

from sklearn.ensemble import IsolationForest  
  
# Assuming 'Ozone' is the column of interest  
X = linear\_imput[['Ozone']].values  
  
# Create an Isolation Forest model  
# You can adjust the hyperparameters like n\_estimators, contamination, etc.  
model = IsolationForest(n\_estimators=10000, contamination=0.05, random\_state=42)  
  
# Fit the model to your data  
model.fit(X)  
  
# Predict outliers (-1 for outliers, 1 for inliers)  
outliers = model.predict(X)  
  
# Plot the data and mark outliers in red  
plt.figure(figsize=(10, 6))  
  
# Plot inliers as a line plot  
plt.plot(df.index, X, color='blue', label='Inliers')  
  
# Scatter plot for outliers in red  
plt.scatter(df.index[outliers == -1], X[outliers == -1], color='red', label='Outliers')  
plt.xlabel('Date')  
plt.ylabel('Ozone')  
plt.title('Isolation Forest Outlier Detection')  
plt.legend()  
plt.show()



**Outlier Detection with K-Means**

K-Means is a widely used unsupervised machine learning algorithm, primarily known for its clustering capabilities. However, its lesser-known superpower lies in its ability to expose outliers.

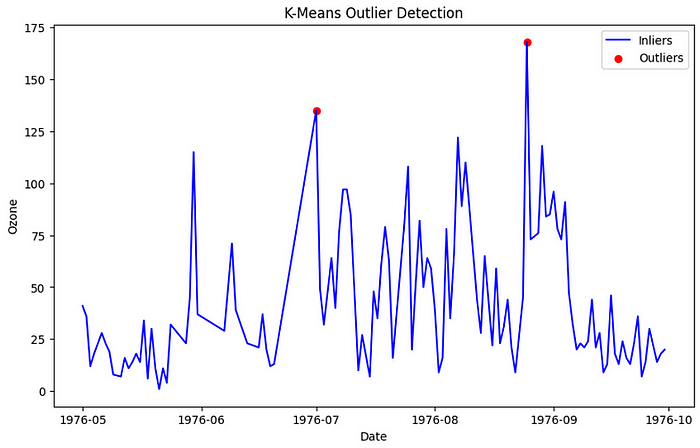
Here’s a glimpse into the inner workings of K-Means for outlier detection:

1. Clustering Data: K-Means partitions data points into clusters based on their similarity. The number of clusters (K) is a user-defined hyperparameter.
2. Cluster Assignments: Each data point is assigned to the cluster whose center (centroid) is closest to it.
3. Distance Calculation: K-Means calculates the distance of each data point from its assigned cluster’s center. This distance represents how well the data point fits within its cluster.
4. Outlier Identification: Outliers often reveal themselves as data points that are far from any cluster’s center. By setting a threshold on these distances, we can pinpoint outliers.

The approach using a fixed percentile threshold in K-Means outlier detection can potentially lead to the identification of outliers, even when there are none present in the data. This method assumes that there are always outliers in the data, which may not be the case.

In practice, it’s essential to consider the characteristics of your dataset and the problem you’re trying to solve.

from sklearn.cluster import KMeans  
  
# Assuming 'Ozone' is the column of interest  
X = linear\_imput[['Ozone']].values  
  
# Create a KMeans model  
# You can adjust the number of clusters (n\_clusters) and other hyperparameters  
n\_clusters = 2 # Adjust as needed  
model = KMeans(n\_clusters=n\_clusters, random\_state=42)  
  
# Fit the model to your data  
model.fit(X)  
  
# Assign each data point to a cluster  
cluster\_assignments = model.predict(X)  
  
# Calculate the distance of each data point to its cluster center  
cluster\_centers = model.cluster\_centers\_  
distances = np.linalg.norm(X - cluster\_centers[cluster\_assignments], axis=1)  
  
# Define a threshold to identify outliers  
threshold = np.percentile(distances, 99) # Adjust as needed  
  
# Identify outlier indices  
outlier\_indices = np.where(distances > threshold)[0]  
  
# Plot the data with outliers marked in red  
plt.figure(figsize=(10, 6))  
plt.plot(df.index, X, c='blue', label='Inliers')  
plt.scatter(df.index[outlier\_indices], X[outlier\_indices], c='red', label='Outliers')  
plt.xlabel('Date')  
plt.ylabel('Ozone')  
plt.title('K-Means Outlier Detection')  
plt.legend()  
plt.show()



**Outlier Detection Local Outlier Factor (LOF).**

In the realm of anomaly detection, Local Outlier Factor (LOF) is a powerful and widely-used algorithm. It excels in identifying anomalies within datasets, especially when dealing with complex, high-dimensional data. LOF stands out for its ability to detect local outliers — anomalies that deviate from their local neighborhood rather than from the dataset as a whole.

Concepts:

1. Local Density: LOF is based on the concept of local density. It calculates the density of data points within a specified neighborhood around a target point. High-density regions are considered normal, while low-density regions are suspicious and may contain outliers.
2. Local Reachability Density: LOF goes further by introducing the concept of local reachability density. This metric measures how reachable a point is from its neighbors. A point is considered an outlier if it has significantly lower reachability density compared to its neighbors.

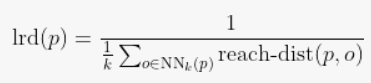
* **k-Distance (k-distance(p)):**This is the distance between a data point ‘p’ and its k-th nearest neighbor. It’s defined as:



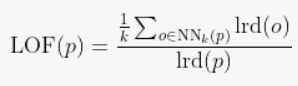
* **Reachability Distance (reach-dist(p, o))**:  
  This metric calculates the reachability distance from point ‘p’ to a neighboring point ‘o’. It’s defined as:



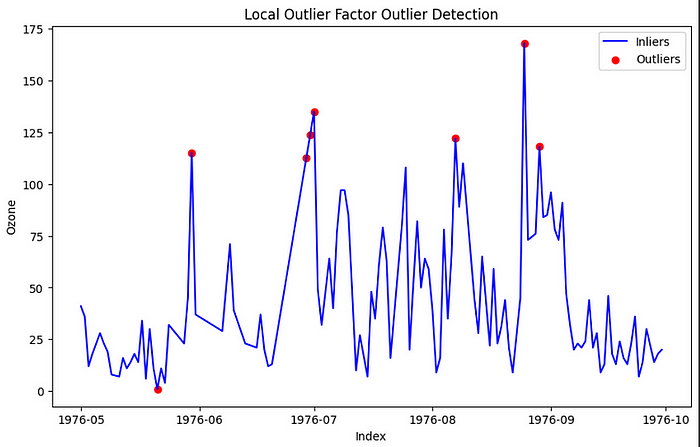
* **Local Reachability Density (lrd(p))**:  
  This density measures how densely a point is surrounded by its neighbors. It’s computed as the inverse of the average reachability distance from point ‘p’ to its k-nearest neighbors:



* **Local Outlier Factor (LOF)**:  
  Finally, LOF quantifies how much an object deviates from its local neighborhood’s density. It’s defined as the ratio of the average local reachability density of a point’s neighbors to its own local reachability density:



from sklearn.neighbors import LocalOutlierFactor  
  
# Assuming 'Ozone' is the column of interest  
X = linear\_imput[['Ozone']].values  
  
# Create a Local Outlier Factor model  
# You can adjust the hyperparameters like n\_neighbors, contamination, etc.  
model = LocalOutlierFactor(n\_neighbors=20, contamination=0.05)  
  
# Fit the model to your data and predict outlier scores  
outlier\_scores = model.fit\_predict(X)  
  
# Plot the data and mark outliers in red  
plt.figure(figsize=(10, 6))  
  
# Plot inliers as a line plot  
plt.plot(df.index, X, color='blue', label='Inliers')  
  
# Scatter plot for outliers in red  
plt.scatter(df.index[outlier\_scores == -1], X[outlier\_scores == -1], color='red', label='Outliers')  
plt.xlabel('Date')  
plt.ylabel('Ozone')  
plt.title('Local Outlier Factor Outlier Detection')  
plt.legend()  
plt.show()



Outliers, those exceptional data points that deviate from the norm, pose intriguing challenges and opportunities in data analysis. To handle them effectively, one must adopt a thoughtful and context-aware approach. Here’s a summary of key considerations:

1. **Understand the Context:**  
   Before taking action, delve into the nature and origin of outliers. Are they genuine insights or data errors? Understanding their context is crucial.
2. **Preserve Valuable Insights:**Outliers can carry valuable insights or information. Consider leaving them untouched if they provide meaningful insights.
3. **Data Transformation:**  
   Apply data transformations or scaling to reduce the impact of outliers while preserving their presence in the dataset.
4. **Removal with Caution:**  
   Removing outliers should be done sparingly, with strong justification. Removing outliers can alter the dataset’s distribution, potentially skewing results or leading to inaccurate conclusions in your analysis.
5. **Imputation Techniques:**  
   Replace outliers with imputed values using data imputation techniques, similar to how one would handle missing data.
6. **Separate Analysis:**  
   In some cases, conducting separate analyses with and without outliers can provide deeper insights into their impact.
7. **Robust Models:**  
   Consider using modeling techniques that are robust to outliers, especially in predictive modeling tasks.
8. **Document and Justify:**  
   Always document your outlier-handling decisions, providing clear reasoning for the chosen approach. Transparency is key.

Ultimately, the handling of outliers should align with your specific analysis goals and the characteristics of your dataset. It’s a nuanced process that requires thoughtful consideration of the specific context in which you are working. Outliers can be enigmatic, but with precision and care, they can lead to discoveries that enrich your data analysis journey.

**Filters on Time Series Analysis**

The analysis of time series data is essential in many fileds such as finance, economics, engineering, and environmental science among others. At its core, a time series is a sequence of data points, indexed (or listed or graphed) in time order. However, real-world time series data is rarely clean; it often containts noise that can obscure underlying patterns and trends. This is where filters and noise reduction techniques come into play.

Before diving into the techniques used to filter out noise, it’s essential to understand what we mean by “noise”. In the context of time series data, noise refers to randomfluctuations that are not part of the underlying trend or periodicity of the data. These fluctuations can arise from varios sources, such as sensor errors, recording errors, or inherent variability in the system being meassured. While some level of noise is inevitable in any data collection process, excessive noise can hinder our ability to extract meaningful insights from the data.

**Filter Classification**

In order to clean up and get a tidy time series data we can use several techniques known as filters. While any of those works in the frequecy domain of the signal (Figure 1), filters can be classified in three different groups: Low-Pass, High-Pass, and Band-Pass.

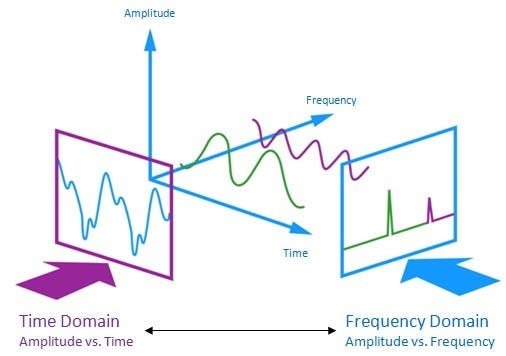


Fig 1. Time-Frequency Domain Analysis

**Low-Pass filters** attenuates the higher frequencies of the signal, allowing the low frequencies to pass without alterations. These filters are used to smoothen the time series and filter out the high frequency noise attributed to all the measure errors explained above. A comon example of this category is the *Simple Moving Average Filter (SMA)*.

**High-Pass filters**attenuates the lower frequencies of the signal, allowing the higher frequencies to pass without alteration and are commonly used to highlight the short term variations of the time series. An example of this category is the *Diferential filter*.

Last but not least are the **Band-Pass filters**, these categories filters out an specific range of frequencies from the time series and are commonly used for filtering the interference in telecommunications as the electric noise, or line frequencies. An exmaple of this group is the *Notch Filter* or *Stop Band filter*.

**Dataset**

In order to illustrate some examples we used the EEG Dataset about a Electroencephalogram signal from [Kaggle](https://www.kaggle.com/datasets/jbouv27/eeg).

import numpy as np # linear algebra  
import pandas as pd # data processing, CSV file I/O (e.g. pd.read\_csv)  
import matplotlib.pyplot as plt  
%matplotlib inline  
  
df=pd.read\_csv("/content/extrait\_wSleepPage01.csv", delimiter=";", thousands=',')  
df.columns  
df['EOG Left']=df['EOG Left'].astype(float)  
df = df[:][:400]  
plt.figure(figsize=(10,8))  
plt.plot(df['EEG C3-A1'])  
  
df.head()

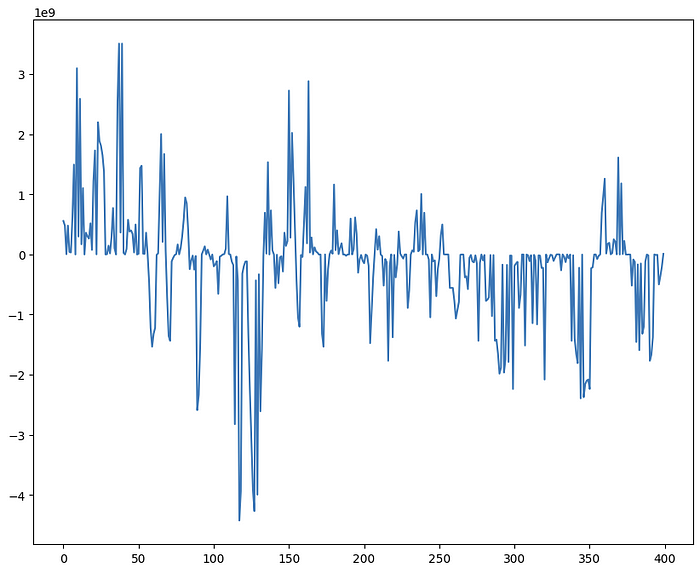


Fig 2. Original Time Series from EEG Dataset.

**Low-Pass Filters**

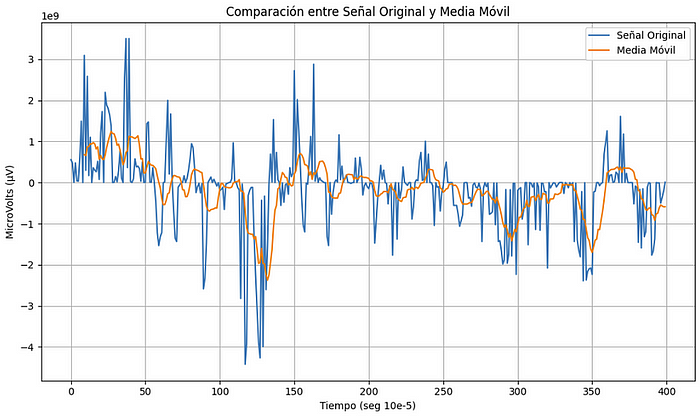
One of the most common filters for noise reduction in time series is the **Simple Moving Average Filter**. This filter works by creating a mean that moves through different time intervals, smoothing out short-term variation and removing noise. The moving average is calculated by taking the average of a number of adjacent observations, known as the time window.

**SMA=xi+xi+1+…+xi+nn−i**

where:  
xi is the first observation,  
xn is the last observation,  
n−i is the time window (number of observations).

The size of the time window is adjusted about the frequency of the noise present in the data, e. g. if the noise is high frequency, we’ll adjust the time window smaller, on the other hand if the noise is low frequency, we’ll adjust a bigger time window.

# Función para calcular la media móvil utilizando el   
# método .rolling() de la libreria pandas.  
  
def media\_movil(data, ventana):  
 return data.rolling(window=ventana).mean()  
  
# Aplicar media móvil con una ventana de 10  
df['Media\_Movil'] = media\_movil(df['EEG C3-A1'], 10)  
  
#print(df)  
  
# Gráfica  
plt.figure(figsize=(10, 6))  
plt.plot(df['EEG C3-A1'], label='Señal Original')  
plt.plot(df['Media\_Movil'], label='Media Móvil')  
plt.title('Comparación entre Señal Original y Media Móvil')  
plt.xlabel('Tiempo (seg 10e-5)')  
plt.ylabel('MicroVolts (μV)')  
plt.legend()  
plt.grid(True)  
plt.tight\_layout()  
plt.show()



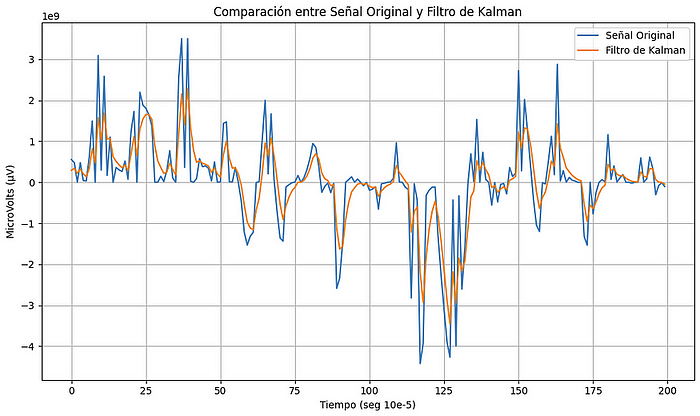
Another commonly used filter is the **Kalman Filter**. This filter uses a mathematical model to estimate the hidden state of a system from the observations made. In the case of time series, the hidden state is the trend of the series and the observations are the observed values of the series. The Kalman filter estimates the trend and noise in the time series and combines them to produce a smoothed time series.

This filter uses a recursive algorithm that is divided into two components, the prediction and the correction.

The first stage (Prediction) performs an estimate of the prior value and calculates the covariance of the error associated with the prior estimate.

For the second stage (Correction), the measurement residual is updated using the current value and subtracting the relationship between the current measurement and the previous estimate, the variable called Kalman gain, the posterior estimate, and the associated covariance are calculated. to the posterior estimation.

!pip install pykalman  
  
from pykalman import KalmanFilter  
import matplotlib.pyplot as plt  
import numpy as np  
  
# Función para aplicar el filtro de Kalman  
def kalman\_filter(data):  
  
 # Suposiciones iniciales para el filtro  
 kf = KalmanFilter(initial\_state\_mean=0, n\_dim\_obs=1)  
  
 # Estimar parámetros con los datos  
 kf = kf.em(data, n\_iter=10)  
  
 # Aplicar el filtro a los datos  
 (filtered\_state\_means, \_) = kf.filter(data)  
 return filtered\_state\_means  
  
# Aplicar filtro de Kalman  
df['Kalman'] = kalman\_filter(df['EEG C3-A1'])  
  
# Gráfica  
plt.figure(figsize=(10, 6))  
plt.plot(df['EEG C3-A1'][:200], label='Señal Original')  
plt.plot(df['Kalman'][:200], label='Filtro de Kalman')  
plt.title('Comparación entre Señal Original y Filtro de Kalman')  
plt.xlabel('Tiempo (seg 10e-5)')  
plt.ylabel('MicroVolts (μV)')  
plt.legend()  
plt.grid(True)  
plt.tight\_layout()  
plt.show()



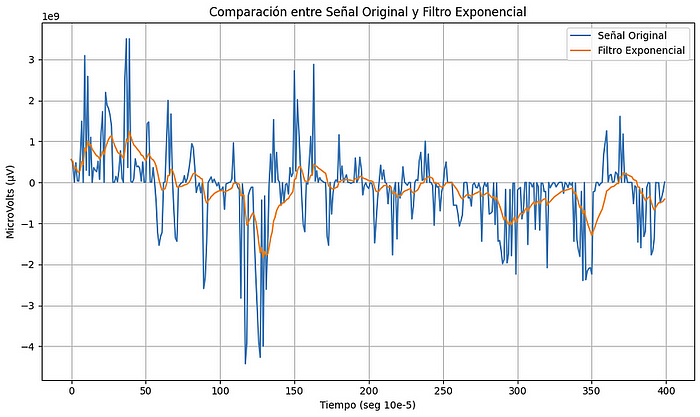
The **Exponential Smoothing Filter** also known as **Exponential Moving Averge (EMA)**, is implemented by using an exponential weighting function to average the signal values at different times. The exponential weighting function assigns a higher weight to the most recent values of the signal and a lower weight to the oldest values. The smoothing constant is a filter parameter that controls the rate of decay of the exponential weights over time.

It is important to mention that the exponential filter is a type of moving average filter, but unlike traditional moving average filters, the exponential filter does not use a fixed size window to average the signal values. Instead, it uses an exponential weighting function to assign weights to signal values at different times, where the ‘α’ parameter controls how quickly changes in the underlying values are responded to.

EMA=∑αixi+EMAi−1αi−1

where:  
xi = Current observation,  
α = 21+N , Smoothing factor,  
N = Number of periods

import matplotlib.pyplot as plt  
  
# Función para aplicar el filtro exponencial (media móvil exponencial)  
def filtro\_exponencial(data, alpha):  
 return data.ewm(alpha=alpha).mean()  
  
# Aplicar filtro exponencial con un factor de suavizado alpha  
alpha = 0.1  
df['Exponencial'] = filtro\_exponencial(df['EEG C3-A1'], alpha)  
  
# Gráfica  
plt.figure(figsize=(10, 6))  
plt.plot(df['EEG C3-A1'], label='Señal Original')  
plt.plot(df['Exponencial'], label='Filtro Exponencial')  
plt.title('Comparación entre Señal Original y Filtro Exponencial')  
plt.xlabel('Tiempo (seg 10e-5)')  
plt.ylabel('MicroVolts (μV)')  
plt.legend()  
plt.grid(True)  
plt.tight\_layout()  
plt.show()



The **Kernel Filter** for time series is a non-parametric smoothing technique. Instead of imposing a specific model on the data (as in ARIMA models), kernel filtering uses weights based on a kernel function to smooth the time series. This type of filter is similar to a moving average, but with weights that change depending on the chosen kernel function.

Generally speaking, kernel filtering computes a weighted average of the points in a time series, where the weights are determined based on the distance between the center point (the point being smoothed) and the surrounding points. The function that defines these weights is the kernel function.

Various kernel functions can be used, including:

* ***Gaussian Kernel***
* ***Kernel Epanechnikov***
* ***triangular kernel***
* ***uniform kernel***
* ***Among others***

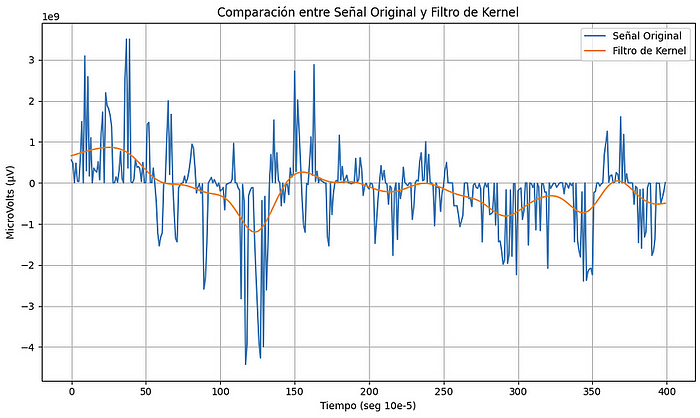
The degree of smoothing is generally controlled by a parameter called “*bandwidth*”. A large bandwidth will result in stronger smoothing, while a small bandwidth will result in lighter smoothing.

The general formula for kernel smoothing is:

**y^(t)=∑Ni=1ωiy(ti)**

where:  
y^(t) is the smoothed value at time t,  
y(ti) are the original values of the time series,  
ωi are the weights calculated from the kernel function.

import matplotlib.pyplot as plt  
import numpy as np  
from statsmodels.nonparametric.kernel\_regression import KernelReg  
  
# Función para aplicar el filtro de kernel  
def filtro\_kernel(data, bandwidth):  
 # Ajustar el modelo de regresión con kernel  
 kr = KernelReg(endog=data, exog=np.arange(len(data)), var\_type='c', reg\_type='lc', bw=[bandwidth])  
 mean, \_ = kr.fit(np.arange(len(data)))  
 return mean  
  
# Aplicar filtro de kernel  
bandwidth = 10 # Ajusta este valor según tus necesidades  
df['Kernel'] = filtro\_kernel(df['EEG C3-A1'], bandwidth)  
  
# Gráfica  
plt.figure(figsize=(10, 6))  
plt.plot(df['EEG C3-A1'], label='Señal Original')  
plt.plot(df['Kernel'], label='Filtro de Kernel')  
plt.title('Comparación entre Señal Original y Filtro de Kernel')  
plt.xlabel('Tiempo (seg 10e-5)')  
plt.ylabel('MicroVolts (μV)')  
plt.legend()  
plt.grid(True)  
plt.tight\_layout()  
plt.show()



**High-Pass Filters**

The **Differential Filter** is a simple but effective technique that acts like a high pass filter. As its name suggests, it is based on signal differentiation. When differentiating a signal, rapid changes in the signal, typically corresponding to high-frequency components, are highlighted, while low-frequency components (such as slow or steady trends) are attenuated.

The differentiation operation in the time domain corresponds to a multiplication by jω (where j is the imaginary unit and ω is the angular frequency) in the frequency domain. This means that higher frequencies are amplified more than lower frequencies, resulting in a high pass filter effect.

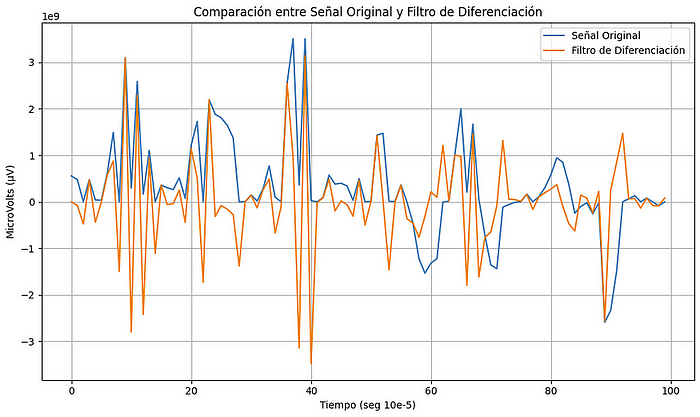
In the context of time series, differentiation is typically done as a discrete difference between consecutive data points:

y(t)=x(t)−x(t−1)

where:  
x(t) is the original value at time t ,  
y(t) is the differenced value at time t .

Although the differential filter is a useful tool, it also has drawbacks. One of the main ones is that it can amplify noise. Since noise tends to have high-frequency components, differentiation can make the noise more prominent in the filtered signal.

import matplotlib.pyplot as plt  
import numpy as np  
  
# Función para aplicar el filtro de diferenciación  
def filtro\_diferenciacion(data):  
 return np.diff(data, prepend=data[0]) # prepend se utiliza para mantener la misma longitud  
  
# Aplicar filtro de diferenciación  
df['Diferenciacion'] = filtro\_diferenciacion(df['EEG C3-A1'])  
  
# Gráfica  
plt.figure(figsize=(10, 6))  
plt.plot(df['EEG C3-A1'][:100], label='Señal Original')  
plt.plot(df['Diferenciacion'][:100], label='Filtro de Diferenciación')  
plt.title('Comparación entre Señal Original y Filtro de Diferenciación')  
plt.xlabel('Tiempo (seg 10e-5)')  
plt.ylabel('MicroVolts (μV)')  
plt.legend()  
plt.grid(True)  
plt.tight\_layout()  
plt.show()



The**Sobel Filter**is a convolution operator that is mainly used in image processing for edge detection. This operator detects edges by calculating the gradient approximation of the intensity of an image at each point.

Edge detection is essentially a high-pass operation, since the edges in an image represent rapid transitions in intensity, or abrupt changes, which are high-frequency components.

The Sobel operator uses two 3×3 convolution masks (or kernels), one to detect changes in the horizontal direction (vertical edges) and one to detect changes in the vertical direction (horizontal edges).

The convolution matrices for the Sobel filter are:

For detection in the horizontal direction Gx :

[−1 −2 −1]

[0 0 0]

[1 2 1]

For detection in the vertical direction  
[−1 0 1]

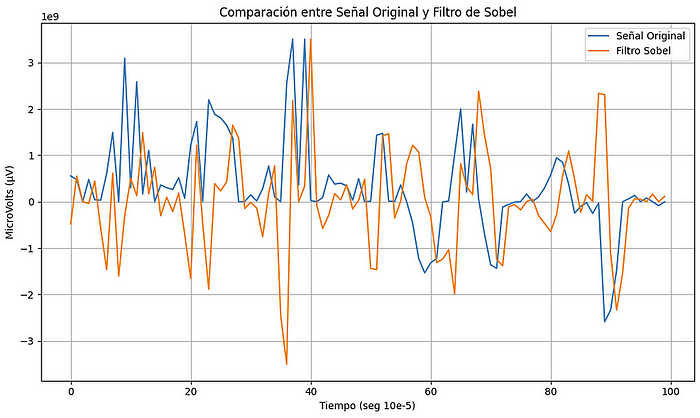
[−2 0 2]

[−1 0 1]

When applying the Sobel filter to an image, areas of rapid intensity transition are highlighted, indicating the presence of an edge. Areas with slow or no change transitions remain dark. This results in an image that emphasizes the edges and discards much of the low frequency information.

Although the Sobel filter is mainly used in image processing, the underlying concept of abrupt change detection can be applied to time series. For example, if you have a time series and want to identify points where there are rapid changes (similar to image edge detection), you might consider adapting and applying an operator similar to Sobel’s. However, in practice, there are other techniques and filters that are often more common for this purpose in the context of time series.

import matplotlib.pyplot as plt  
import numpy as np  
  
# Función para aplicar el "filtro Sobel" a la serie temporal  
def filtro\_sobel(data):  
 # Sobel Kernel para una dimensión  
 kernel = np.array([-1, 0, 1])  
 return np.convolve(data, kernel, mode='same') # 'same' para mantener la misma longitud  
  
# Aplicar filtro Sobel  
df['Sobel'] = filtro\_sobel(df['EEG C3-A1'])  
  
# Gráfica  
plt.figure(figsize=(10, 6))  
plt.plot(df['EEG C3-A1'][:100], label='Señal Original')  
plt.plot(df['Sobel'][:100], label='Filtro Sobel')  
plt.title('Comparación entre Señal Original y Filtro de Sobel')  
plt.xlabel('Tiempo (seg 10e-5)')  
plt.ylabel('MicroVolts (μV)')  
plt.legend()  
plt.grid(True)  
plt.tight\_layout()  
  
plt.show()



The **FIR Filters (Finite Impulse Response)** are a type of digital filter whose impulse response has a finite duration, that is, it is set to zero at some finite point in time. This feature is in contrast to IIR filters, where the response can, in theory, continue indefinitely.

This type of filter can be implemented using Hamming, Blackman, or Hann window designs, among others.

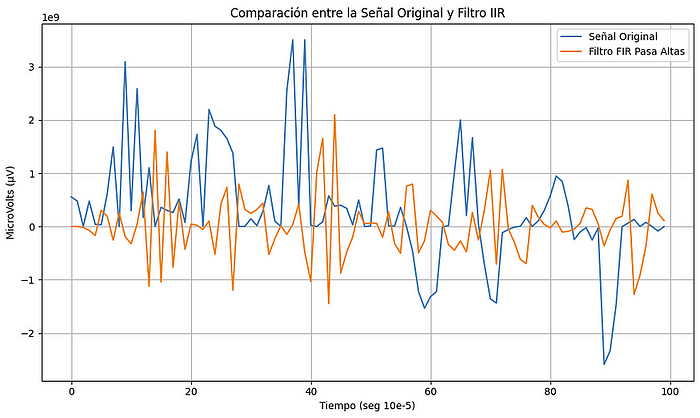
The Hamming window has a particular form defined by the equation:

**ω(n)=0.54−0.46cos(2πnN−1)**

where:  
ω(n) is the value of the window at point n,  
N is the total number of points in the window.

These types of filters are generally replaced by IIR filters due to the delay generated. Since FIR filters are non-recursive, their impulse response has an inherent delay, which is about half the length of the filter. In real-time applications or where latency is critical, this delay can be a limiting factor.

!pip install scipy  
import matplotlib.pyplot as plt  
import numpy as np  
from scipy.signal import firwin, lfilter  
  
# Función para aplicar el filtro FIR pasa altas  
def filtro\_fir\_pasa\_altas(data, order, cutoff\_freq):  
 # Diseño del filtro FIR pasa altas usando una ventana de Hamming  
 coeffs = firwin(order + 1, cutoff\_freq, pass\_zero=False, window='hamming')  
 return lfilter(coeffs, 1.0, data)  
  
# Parámetros para el filtro FIR  
order = 10  
cutoff\_freq\_normalized = 0.25 # Ajusta según tus necesidades.  
  
# Aplicar filtro FIR pasa altas  
df['FIR\_Pasa\_Altas'] = filtro\_fir\_pasa\_altas(df['EEG C3-A1'], order, cutoff\_freq\_normalized)  
  
# Gráfica  
plt.figure(figsize=(10, 6))  
plt.plot(df['EEG C3-A1'][:100], label='Señal Original')  
plt.plot(df['FIR\_Pasa\_Altas'][:100], label='Filtro FIR Pasa Altas')  
plt.title('Comparación entre la Señal Original y Filtro IIR')  
plt.xlabel('Tiempo (seg 10e-5)')  
plt.ylabel('MicroVolts (μV)')  
plt.legend()  
plt.grid(True)  
plt.tight\_layout()  
  
plt.show()



The **IIR (Infinite Impulse Response) Filters** are digital filters that have an impulse response that extends indefinitely, theoretically to infinity. These filters are characterized by their recursive structure, where the output depends not only on current and past inputs but also on past outputs. This feature is in contrast to FIR (Finite Impulse Response) filters, where the output depends only on current and past inputs.

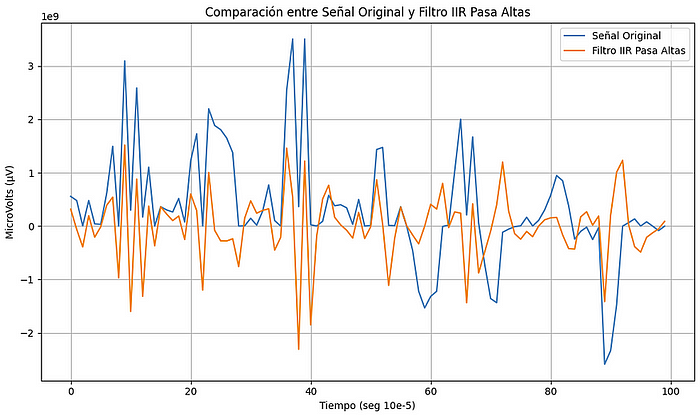
This type of filter can be implemented using a Butterworth, Chevishev, or elliptical design, depending on the application.

The cutoff frequency, using a Butterworth filter design, must be normalized with the sampling frequency of the data:

**fcnorm=fc/fs**

where:  
fcnorm is the normalized cutoff frequency,  
fc is the desired cutoff frequency,  
fs is the sampling frequency of the data.

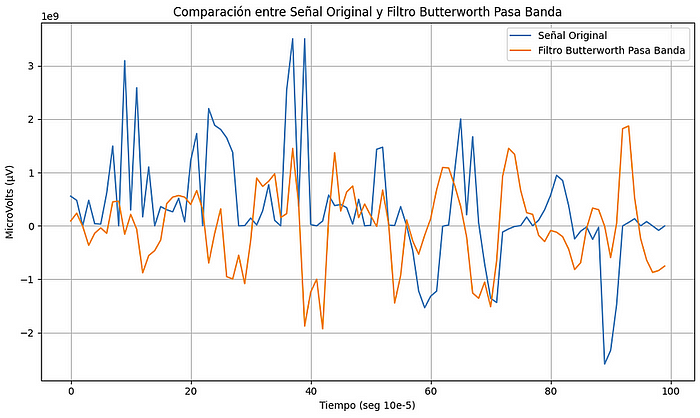
import matplotlib.pyplot as plt  
import numpy as np  
from scipy.signal import butter, lfilter  
  
# Nota: Asumo que df['EEG C3-A1'] y las importaciones necesarias ya están definidos en tu código.  
  
# Función para aplicar el filtro IIR pasa altas  
# Frecuencia de muestreo 200Hz  
# Frecuencia de corte 50Hz  
def filtro\_iir\_pasa\_altas(data, order=2, cutoff\_freq=0.25):  
 # Diseño del filtro IIR pasa altas usando Butterworth  
 b, a = butter(order, cutoff\_freq, btype='high', analog=False)  
 return lfilter(b, a, data)  
  
# Aplicar filtro IIR pasa altas  
df['IIR\_Pasa\_Altas'] = filtro\_iir\_pasa\_altas(df['EEG C3-A1'])  
  
# Gráfica  
plt.figure(figsize=(10, 6))  
plt.plot(df['EEG C3-A1'][:100], label='Señal Original')  
plt.plot(df['IIR\_Pasa\_Altas'][:100], label='Filtro IIR Pasa Altas')  
plt.title('Comparación entre Señal Original y Filtro IIR Pasa Altas')  
plt.xlabel('Tiempo (seg 10e-5)')  
plt.ylabel('MicroVolts (μV)')  
plt.legend()  
plt.grid(True)  
plt.tight\_layout()  
  
plt.show()



**Band-Pass Filters**

**Butterworth Filters**, also known as maximally flat filters, are famous for their flat frequency response in the passband. That is, they have no ripples or ripples in the passband or stopband, resulting in a smooth, monotonic response.

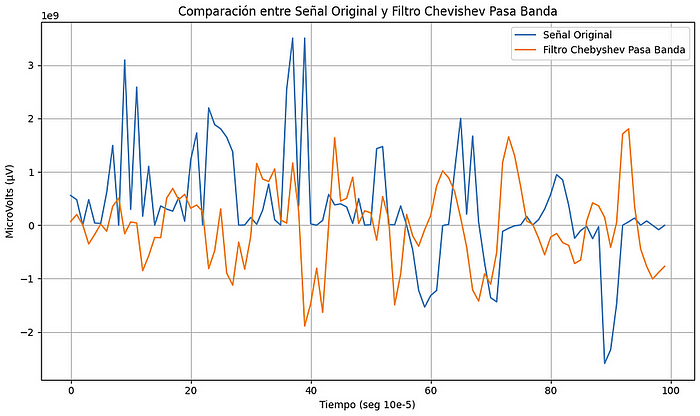
import matplotlib.pyplot as plt  
import numpy as np  
from scipy.signal import butter, lfilter  
  
# Función para aplicar el filtro Butterworth Pasa Banda  
def filtro\_butterworth\_pasa\_banda(data, lowcut, highcut, fs, order):  
 nyq = 0.5 \* fs  
 low = lowcut / nyq  
 high = highcut / nyq  
 b, a = butter(order, [low, high], btype='band')  
 return lfilter(b, a, data)  
  
# Frecuencias de corte  
lowcut = 10.0 # Frecuencia de corte inferior  
highcut = 70.0 # Frecuencia de corte superior  
  
# Aplicar filtro Butterworth Pasa Banda  
df['Butterworth\_Pasa\_Banda'] = filtro\_butterworth\_pasa\_banda(df['EEG C3-A1'], lowcut, highcut, fs=200, order=4)  
  
# Gráfica  
plt.figure(figsize=(10, 6))  
plt.plot(df['EEG C3-A1'][:100], label='Señal Original')  
plt.plot(df['Butterworth\_Pasa\_Banda'][:100], label='Filtro Butterworth Pasa Banda')  
plt.title('Comparación entre Señal Original y Filtro Butterworth Pasa Banda')  
plt.xlabel('Tiempo (seg 10e-5)')  
plt.ylabel('MicroVolts (μV)')  
plt.legend()  
plt.grid(True)  
plt.tight\_layout()  
plt.show()



**Chebyshev Filters**, also known for their equirripple response (equidistant ripple), can be designed to have ripples within the passband or stopband, depending on the type of Chebyshev filter being used. Chebyshev Type I filters have passband ripples, while Chebyshev Type II filters have stopband ripples.

Unlike Butterworth filters, which have a flat passband response, Chebyshev filters have ripples. These ripples allow for a faster transition between the passband and stopband, at the cost of a less flat response.

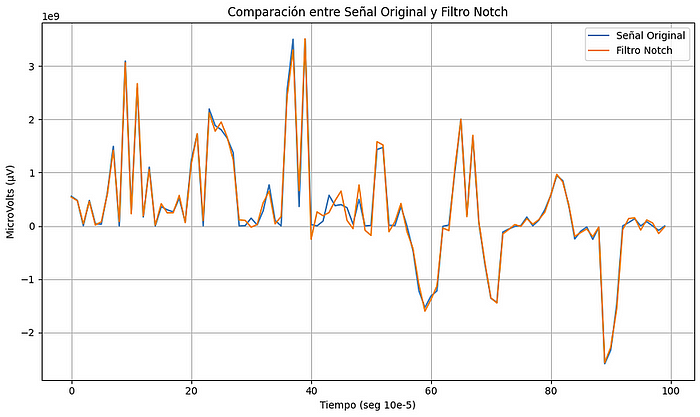
import matplotlib.pyplot as plt  
import numpy as np  
from scipy.signal import cheby1, lfilter  
  
# Función para aplicar el filtro Chebyshev Tipo I Pasa Banda  
def filtro\_chebyshev\_pasa\_banda(data, lowcut, highcut, fs, order, ripple\_db):  
 nyq = 0.5 \* fs  
 low = lowcut / nyq  
 high = highcut / nyq  
 b, a = cheby1(order, ripple\_db, [low, high], btype='band')  
 return lfilter(b, a, data)  
  
# Frecuencias de corte y ondulación  
lowcut = 10.0 # Frecuencia de corte inferior  
highcut = 70.0 # Frecuencia de corte superior  
ripple\_db = 0.5 # Ondulación en dB en la banda de paso  
  
# Aplicar filtro Chebyshev Tipo I Pasa Banda  
df['Chebyshev\_Pasa\_Banda'] = filtro\_chebyshev\_pasa\_banda(df['EEG C3-A1'], lowcut, highcut, fs=200, order=4, ripple\_db=ripple\_db)  
  
# Gráfica  
plt.figure(figsize=(10, 6))  
plt.plot(df['EEG C3-A1'][:100], label='Señal Original')  
plt.plot(df['Chebyshev\_Pasa\_Banda'][:100], label='Filtro Chebyshev Pasa Banda')  
plt.title('Comparación entre Señal Original y Filtro Chevishev Pasa Banda')  
plt.xlabel('Tiempo (seg 10e-5)')  
plt.ylabel('MicroVolts (μV)')  
plt.legend()  
plt.grid(True)  
plt.tight\_layout()  
plt.show()



A **notch filter**, also known as a “trap” filter, is designed to remove a specific frequency or a narrow range of frequencies from a signal. Essentially, it “punches a hole” (hence the name “notch”) in the signal’s frequency spectrum in the specified range, allowing frequencies above and below the rejection range to pass through with little disturbance.

In general, one of the limitations of the notch filter is that they handle a very narrow bandwidth.

import matplotlib.pyplot as plt  
import numpy as np  
from scipy.signal import iirnotch, lfilter  
  
# Nota: Asumo que df['EEG C3-A1'] y las importaciones necesarias ya están definidos en tu código.  
  
# Función para aplicar el filtro Notch  
def filtro\_notch(data, f0, Q=30, fs=200): # asumiendo una frecuencia de muestreo de 200 Hz  
 b, a = iirnotch(f0, Q, fs)  
 return lfilter(b, a, data)  
  
# Frecuencia que queremos eliminar  
f0 = 60.0 # por ejemplo, para eliminar la interferencia de 60 Hz  
  
# Aplicar filtro Notch  
df['Notch\_Filtered'] = filtro\_notch(df['EEG C3-A1'], f0)  
  
# Gráfica  
plt.figure(figsize=(10, 6))  
plt.plot(df['EEG C3-A1'][:100], label='Señal Original')  
plt.plot(df['Notch\_Filtered'][:100], label='Filtro Notch')  
plt.title('Comparación entre Señal Original y Filtro Notch')  
plt.xlabel('Tiempo (seg 10e-5)')  
plt.ylabel('MicroVolts (μV)')  
plt.legend()  
plt.grid(True)  
plt.tight\_layout()  
plt.show()



**Summarizing**

The different types of filters are powerful tools for time series analysis, allowing you to obtain clearer visualizations of the characteristics you want to observe (either long-term trends, or rapid fluctuations of the signals), but it is always necessary to understand the characteristics. capabilities, advantages and disadvantages of each one to get the most out of them.

Low-pass filters allow us to detect long-term trends or very slow and gradual changes within our data, thereby reducing rapid variations in the data as well as noise.

High pass filters are a great option to analyze abrupt changes in the analyzed signal, such as the rate of change of a currency in the market.

Finally, band pass or band rejection filters are a great option to reduce noise in the signal to be analyzed.

**Upsampling and downsampling**

In time series analysis, downsampling and upsampling are techniques used to modify the temporal resolution of a time series by changing the frequency of observations. These techniques can be useful for various purposes, such as data visualization, data compression, and modeling.

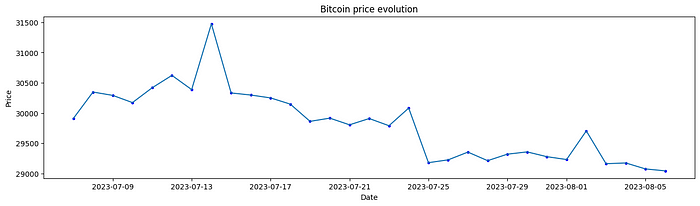
In order to show how these techniques work, we will use an example in python using the data for cryptocurrencies price:

import yfinance as yf  
import pandas as pd  
import matplotlib.pyplot as plt  
  
# Select the our financial asset  
asset = "BTC-USD"  
  
# Specify the range date to get the information  
start\_date = "2023-07-07"  
end\_date = "2023-08-07"  
  
# Downloading the data  
data = yf.download(asset, start=start\_date, end=end\_date)  
  
print(data.head())  
  
print("Total of samples: ", data.index.size)

The first part of this code will download the data that we need.

The code below allows us to plot this info:

plt.figure(figsize=(16, 4))  
plt.xlabel('Date')  
plt.ylabel('Price')  
plt.title('Bitcoin price evolution')  
plt.scatter(data.index, data[['Open']], label=asset, color='b', marker='o', s=8)  
plt.plot(data.index, data[['Open']])



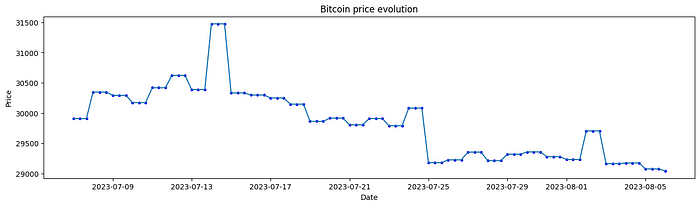
**Upsampling**: It involves increasing the frequency of observations in a time series by interpolating or adding new data points between existing observations. Upsampling can be used to estimate or interpolate values between existing data points.

Upsampling Techniques:

* Linear Interpolation: Estimate values between existing data points using linear interpolation.
* Polynomial Interpolation: Use polynomial equations to estimate values between data points.
* Time-Based Filling: Replicate values from the nearest existing data point to fill in gaps.

data\_upsampled = data[['Open']].resample('8H').ffill()  
  
print(data\_upsampled.head())  
  
print("Total of samples: ", data\_upsampled.index.size)  
  
plt.figure(figsize=(16, 4))  
plt.xlabel('Date')  
plt.ylabel('Price')  
plt.title('Bitcoin price evolution')  
plt.scatter(data\_upsampled.index, data\_upsampled[['Open']], label=asset, color='b', marker='o', s=8)  
plt.plot(data\_upsampled.index, data\_upsampled[['Open']])

As we can see in plot below, the quantity of dots has increased and plot shape has changed.



**Downsampling**: It refers to the process of reducing the frequency of observations in a time series by aggregating or summarizing data over longer time intervals. Downsampling is often used to reduce the volume of data, especially when dealing with high-frequency data. For example, we can aggregate hourly data into daily or monthly summaries.

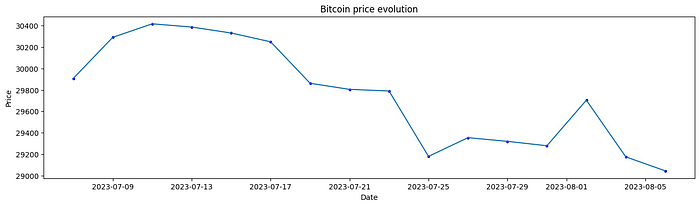
Downsampling Techniques:

* Average Aggregation: Compute the average (or other summary statistic) of data points within each new interval.
* Max or Min Aggregation: Find the maximum or minimum value within each new interval.
* Sum Aggregation: Calculate the sum of data points within each new interval.

We will use the resample function in python to perform and upsampling, code is shown below:

data\_downsampled = data[['Open']].resample('2D').ffill()  
  
print(data\_downsampled.head())  
  
print("Total of samples: ", data\_downsampled.index.size)  
  
plt.figure(figsize=(16, 4))  
plt.xlabel('Date')  
plt.ylabel('Price')  
plt.title('Bitcoin price evolution')  
plt.scatter(data\_downsampled.index, data\_downsampled[['Open']], label=asset, color='b', marker='o', s=8)  
plt.plot(data\_downsampled.index, data\_downsampled[['Open']])

Now the quantity of dots in plot has decreased, in this we took only a few dots from total data, so we had lost information due it was dropped.



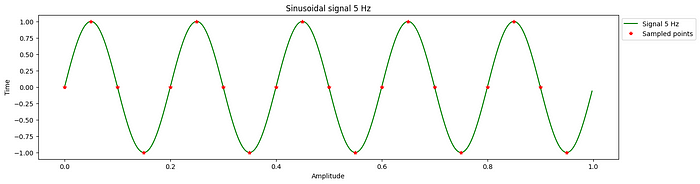
**Aliasing**

Aliasing is an undesirable effect that is seen in sampled systems. When the input frequency is greater than half the sample frequency, the sampled points do not adequately represent the input signal. Inputs at these higher frequencies are observed at a lower, aliased frequency. When we are working with data in time series format it can have same effect if we apply a downsampling to our time series, so we need to be sure we are using the correct method to avoid lost information.

The next example show us what does mean aliasing:

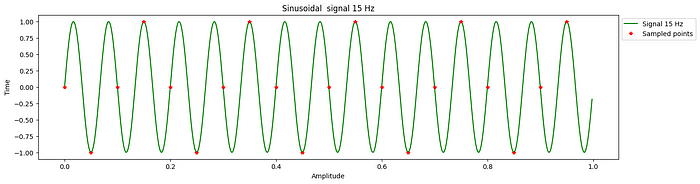
import numpy as np  
  
t = np.arange(0, 1, 1/500)  
ts = np.arange(0, 1, 1/20)  
  
f = 5  
  
#phase for sinewave  
b = np.sin(2\*np.pi\*f\*t)  
  
c = np.sin(2\*np.pi\*f\*ts)  
  
plt.figure(figsize=(16, 4))  
plt.xlabel('Amplitude')  
plt.ylabel('Time')  
plt.title('Sinusoidal signal 5 Hz')  
plt.plot(t,b,"g", label='Signal 5 Hz')  
plt.plot(ts,c,"r\*", label='Sampled points')  
plt.legend(loc='upper left', bbox\_to\_anchor=(1, 1))  
plt.show()

In this case, it’s a sinusoidal signal with a frecuency of 5 Hz, and we have sampled this signal with a frecuency of 20 Hz, due sampled frecuency is greather than 2 times signal frecuency, we don’t have aliasing and original signal can be reconstructed with samples without lost information.

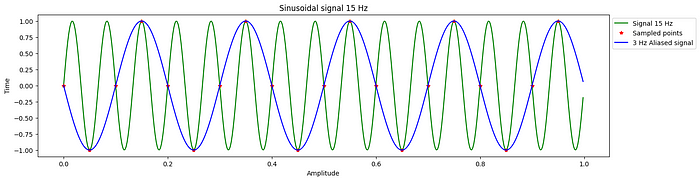


In next case, the signal has a frecuency of 15 Hz and it’s sampled to 20 Hz. So frecuency is smaller than 2 times signal frecuency and aliasing is present.

t = np.arange(0, 1, 1/500)  
ts = np.arange(0, 1, 1/20)  
  
f = 15  
  
#phase for sinewave  
b = np.sin(2\*np.pi\*f\*t)  
  
c = np.sin(2\*np.pi\*f\*ts)  
  
plt.figure(figsize=(16, 4))  
plt.xlabel('Amplitude')  
plt.ylabel('Time')  
plt.title('Sinusoidal signal 15 Hz')  
plt.plot(t,b,"g", label='Signal 15 Hz')  
plt.plot(ts,c,"r\*", label='Sampled points')  
plt.legend(loc='upper left', bbox\_to\_anchor=(1, 1))  
plt.show()



If we try to reconstruct the original signal we will have problems. The plot below show us the signal that we are going to get with sampled point.



**References**

Pandas Development Team. (n.d.). Pandas: Data structures for efficient data manipulation. Retrieved from [https://pandas.pydata.org/](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fpandas.pydata.org%2F)

SciPy community. (n.d.). SciPy: Open-source library for mathematics, science, and engineering. Retrieved from [https://www.scipy.org/](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fwww.scipy.org%2F)

Scikit-learn contributors. (n.d.). Scikit-learn: Machine Learning in Python. Retrieved from [https://scikit-learn.org/](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fscikit-learn.org%2F)

Saturn Cloud. (2023, September 3). Outlier Detection with K-Means Algorithm: A Comprehensive Guide. Saturn Cloud Blog. [[https://saturncloud.io/blog/outlier-detection-with-kmeans-algorithm-a-comprehensive-guide/]](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fsaturncloud.io%2Fblog%2Foutlier-detection-with-kmeans-algorithm-a-comprehensive-guide%2F%255D)

Author(s): Wikipedia contributors, Title: Isolation Forest, URL: [https://en.wikipedia.org/wiki/Isolation\_forest](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fen.wikipedia.org%2Fwiki%2FIsolation_forest), Accessed: September 3, 2023

Author: Cooley, B. D. URL: [https://medium.com/utility-machine-learning/an-introduction-to-imputation-solving-problems-of-missing-and-insufficient-data-1814c90c3545](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fmedium.com%2Futility-machine-learning%2Fan-introduction-to-imputation-solving-problems-of-missing-and-insufficient-data-1814c90c3545)

Author: Milon, I. H. URL: [https://medium.com/gadictos/outlier-detection-a-comprehensive-overview-of-different-methods-49c18304bfad](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Fmedium.com%2Fgadictos%2Foutlier-detection-a-comprehensive-overview-of-different-methods-49c18304bfad)

Author: Wenig, P. URL: [https://towardsdatascience.com/local-outlier-factor-for-anomaly-detection-cc0c770d2ebe](https://colab.research.google.com/corgiredirector?site=https%3A%2F%2Ftowardsdatascience.com%2Flocal-outlier-factor-for-anomaly-detection-cc0c770d2ebe)

Author: Amol Mavuduru, URL:[https://medium.com/utility-machine-learning/an-introduction-to-imputation-solving-problems-of-missing-and-insufficient-data-1814c90c3545](https://towardsdatascience.com/how-to-perform-anomaly-detection-with-the-isolation-forest-algorithm-e8c8372520bc)

*“Time Series Analysis and Its Applications: With R Examples*” de Robert H. Shumway y David S. Stoffer.