Time Series Analisys on Geoespatial Data with Python

Author: João Otavio Nascimento Firigato

email: joaootavionf007@gmail.com

LinkedIn: https://www.linkedin.com/in/jo%C3%A3o-otavio-firigato-4876b3aa/

First instructions:

✅ Access the link to join our private WhatsApp community for students: https://chat.whatsapp.com/EPn27ZgR07lF3e1vnj8FiI

❗ It is important to access the Whatsapp Group to get the Colab Notebooks, as the PDF files are protected from text copying.

Chapter 5 - Dealing with missing values (Data Imputation)

Imputation in statistics refers to the procedure of using alternative values in place of missing data. Missing information can introduce a significant degree of bias, make data processing and analysis more difficult, and reduce efficiency, which are the three main problems it causes. Imputation of missing data is a crucial step in time series analysis to avoid losing information and introducing bias. However, imputation can also introduce uncertainty and error into the data, so it is essential to choose the appropriate method for the data and the goals of the analysis.

We employ imputation because missing data can lead to the following issues:

- Distorts the dataset: Large amounts of missing data can lead to anomalies in the distribution of variables, which can change the relative importance of different categories in the dataset.
- Cannot be worked with most Python ML libraries.
- Impacts on the final model: Missing data can lead to bias in the dataset, which can affect the analysis of the final model.
- Desire to restore the entire dataset: This typically occurs when we do not want to lose any data in our dataset because all of them are crucial.
- It is important to mention that the quality of data for time series forecasting is very important, hence the importance of a robust time series imputation process.

Missing Data Types:

There are three types of missing data;

- MCAR (Missing Completely at Random): This is a case where the probability of missing values in a variable is the same for all samples. For example, respondents in the data collection process decide whether to include house numbers in a survey completely randomly. Here, all missing values are the result of random chance. This is a rare situation.
- MAR (Missing at Random): This is a case where the variable has missing values that are randomly distributed, however, they are related to some values of other variables. For example, if men are more likely to hide their income level, income is MAR. The absence of income data depends on the gender of the respondent.
- MNAR (Missing not at Random): This is a case where the missing data is related to events or factors that are not measured by the researcher. For example, people with high income levels are less likely to answer the question about their income level due to fear of higher taxes. Here, the absence of income data is not random, it is related to the income level, but the income level is not known to us.

While missing values often have a negative connotation in machine learning and analytics, they are not always harmful. In fact, with proper management, these missing values can be turned into valuable features, generating additional insights and improving the predictive capabilities of machine learning models. The ability to deal with missing data typically depends on two key pieces of information: the reason behind the missing values and the most effective method for managing them.

Strategies for dealing with missing data in time series

Let's connect to Drive to get our data:

In [ ]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call dri
ve.mount("/content/drive", force_remount=True).
```

In [ ]:
```
import pandas as pd
import matplotlib.pyplot as plt
```

We have a CSV with LST data from the MODIS satellite:

Link: https://drive.google.com/drive/folders/1Vn4w8WqHgHta7rwJ5gD_WRARo7b1SLAv?usp=sharing

In [ ]:
```
path = '/content/drive/MyDrive/Datasets_TS/Pontos_LST/LST_cana.csv'
```

In [ ]:
```
data = pd.read_csv(path)
```

In [ ]:
```
data = data[data['unique_id'] == 0].copy()
```

In [ ]:
```
data
```

Out[ ]:

	Unnamed: 0	longitude	latitude	LST_Day_1km	datetime	unique_id
0	7	-50.462861	-21.330496	33.39	2022-01-08	0
1	14	-50.462861	-21.330496	37.93	2022-01-15	0
2	20	-50.462861	-21.330496	35.83	2022-01-21	0
3	21	-50.462861	-21.330496	34.49	2022-01-22	0
4	22	-50.462861	-21.330496	34.77	2022-01-23	0
...	...	...	...	...	...	...
165	342	-50.462861	-21.330496	29.15	2022-12-21	0
166	343	-50.462861	-21.330496	29.61	2022-12-22	0
167	344	-50.462861	-21.330496	27.87	2022-12-23	0
168	345	-50.462861	-21.330496	33.23	2022-12-24	0
169	348	-50.462861	-21.330496	24.61	2022-12-27	0

170 rows × 6 columns

In [ ]:
```
data = data.drop(columns=['Unnamed: 0', 'longitude','latitude', 'unique_id'])
```

Let's convert our date column to an index:

In [ ]:
```
data['datetime'] = pd.to_datetime(data['datetime'])
```

In [ ]:
```
data.set_index('datetime', inplace=True)
```

In [ ]:
```
data
```

Out[ ]:

datetime	LST_Day_1km
2022-01-08	33.39
2022-01-15	37.93
2022-01-21	35.83
2022-01-22	34.49
2022-01-23	34.77
...	...
2022-12-21	29.15
2022-12-22	29.61
2022-12-23	27.87
2022-12-24	33.23
2022-12-27	24.61

170 rows × 1 columns

There are some dates where we don't have LST information. First let's create these dates by resampling our dataset:

In [ ]:
```
res_data = data.resample('D').asfreq()
```

In [ ]:
```
res_data
```

Out[ ]:

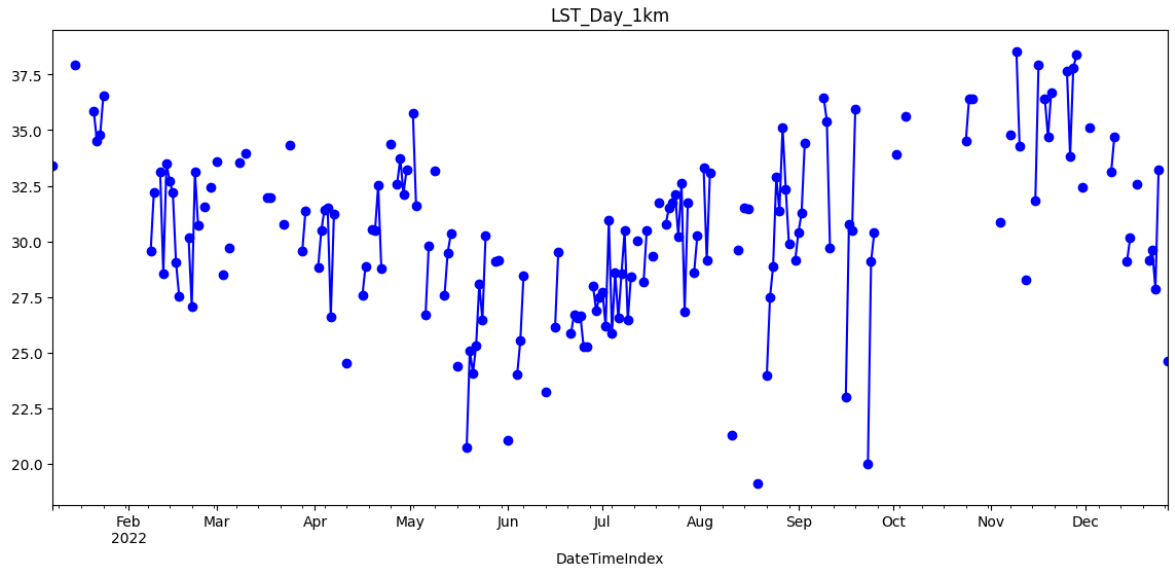
datetime	LST_Day_1km
2022-01-08	33.39
2022-01-09	NaN
2022-01-10	NaN
2022-01-11	NaN
2022-01-12	NaN
...	...
2022-12-23	27.87
2022-12-24	33.23
2022-12-25	NaN
2022-12-26	NaN
2022-12-27	24.61

354 rows × 1 columns

Let's visualize our information with a chart:

In [ ]:
```
sample_data = res_data.copy()
```

In [ ]:
```
sample_data['LST_Day_1km'].plot(title='LST_Day_1km', marker='o', color='blue', f
plt.xlabel('DateTimeIndex')
plt.show()
```



Deletion

This strategy involves eliminating any rows that contain missing values. While simple to perform, if the missing data is not Missing Completely at Random (MCAR), this approach

can result in the loss of valuable information.

In [ ]:
```
sample_data.dropna()
```

Out[ ]:

datetime	LST_Day_1km
2022-01-08	33.39
2022-01-15	37.93
2022-01-21	35.83
2022-01-22	34.49
2022-01-23	34.77
...	...
2022-12-21	29.15
2022-12-22	29.61
2022-12-23	27.87
2022-12-24	33.23
2022-12-27	24.61

170 rows × 1 columns

Constant imputation

This technique replaces all missing values with a constant. Unless there is a compelling reason to select a specific constant, this method is generally not recommended.

In [ ]:
```
sample_data.fillna(0)
```

Out[ ]:

datetime	LST_Day_1km
2022-01-08	33.39
2022-01-09	0.00
2022-01-10	0.00
2022-01-11	0.00
2022-01-12	0.00
...	...
2022-12-23	27.87
2022-12-24	33.23
2022-12-25	0.00
2022-12-26	0.00
2022-12-27	24.61

354 rows × 1 columns

Last Observation Carried Forward (LOCF) and Next Observation Carried Backward (NOCB)

These methods replace missing values with the immediately preceding observed value (LOCF) or the subsequent observed value (NOCB). They are potentially useful for time series data, but may introduce bias if the data are non-stationary.

In [ ]:
```
ffill_data = sample_data.ffill()
ffill_data
```

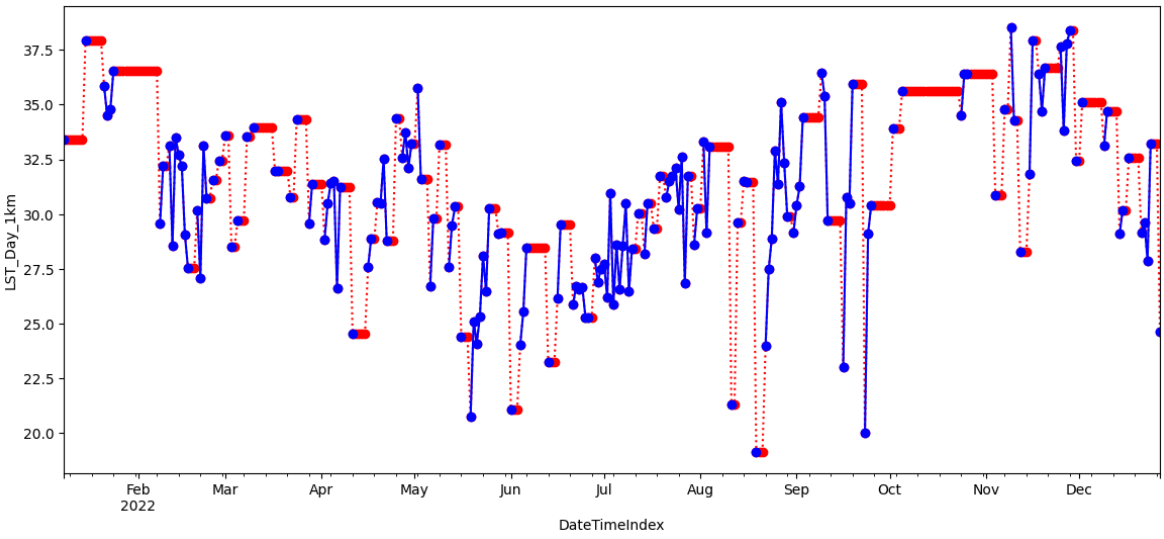
Out[ ]:

datetime	LST_Day_1km
2022-01-08	33.39
2022-01-09	33.39
2022-01-10	33.39
2022-01-11	33.39
2022-01-12	33.39
...	...
2022-12-23	27.87
2022-12-24	33.23
2022-12-25	33.23
2022-12-26	33.23
2022-12-27	24.61

354 rows × 1 columns

In [ ]:
```
ffill_data['LST_Day_1km'].plot(color='red', marker='o', linestyle='dotted', figs

sample_data['LST_Day_1km'].plot(ylabel='LST_Day_1km', marker='o', color='blue',
plt.xlabel('DateTimeIndex')
```

Out[ ]:
```
Text(0.5, 0, 'DateTimeIndex')
```



In [ ]:
```
bfill_data = sample_data.bfill()
bfill_data
```

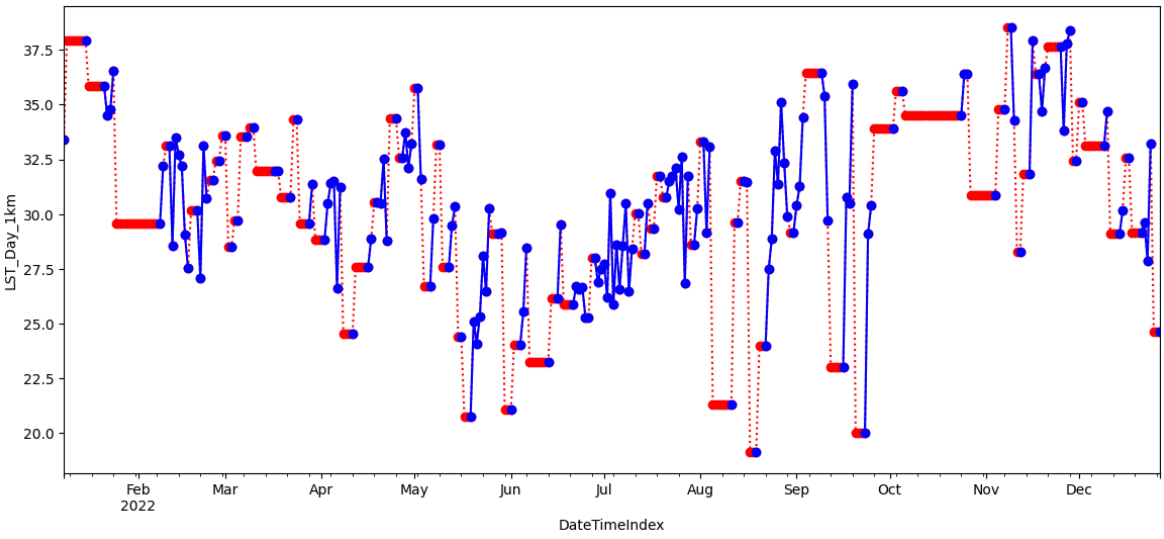
Out[ ]:

datetime	LST_Day_1km
2022-01-08	33.39
2022-01-09	37.93
2022-01-10	37.93
2022-01-11	37.93
2022-01-12	37.93
...	...
2022-12-23	27.87
2022-12-24	33.23
2022-12-25	24.61
2022-12-26	24.61
2022-12-27	24.61

354 rows × 1 columns

In [ ]:
```
bfill_data['LST_Day_1km'].plot(color='red', marker='o', linestyle='dotted', figs

sample_data['LST_Day_1km'].plot(ylabel='LST_Day_1km', marker='o', color='blue',
plt.xlabel('DateTimeIndex')
```

Out[ ]:
```
Text(0.5, 0, 'DateTimeIndex')
```



Mean/Median/Mode

In this approach, missing values are replaced by the mean (for continuous data), median (for ordinal data), or mode (for categorical data) of the available values. Although easy to implement, this method can potentially underestimate the variance.

In [ ]:
```
median_value = sample_data['LST_Day_1km'].median()
```

In [ ]:
```
median_value
```

Out[ ]:
```
30.470000000000027
```

In [ ]:
```
median_data = sample_data.fillna(median_value)
median_data
```

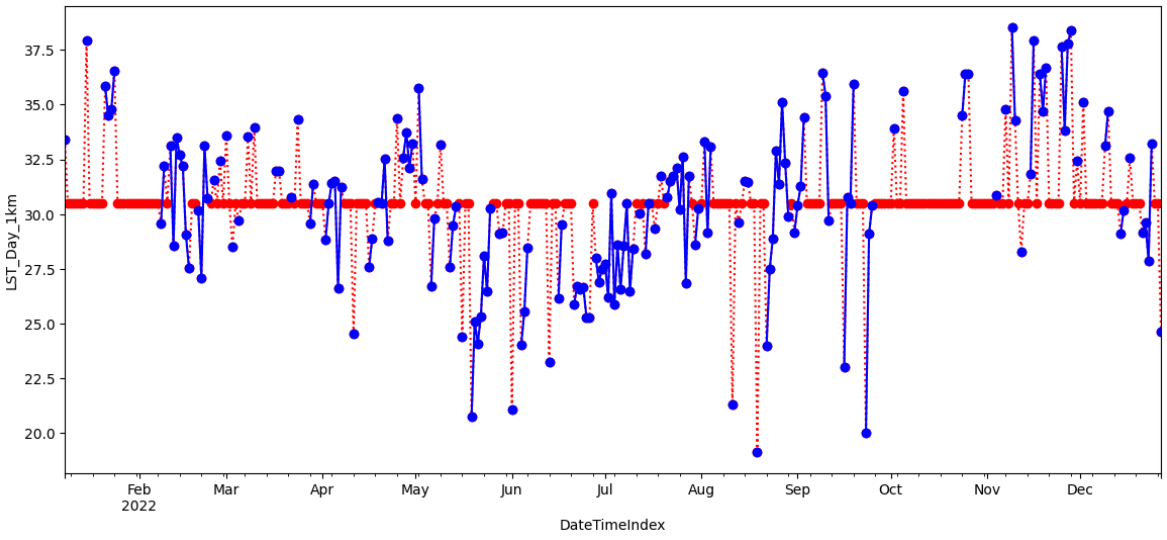
Out[ ]:

datetime	LST_Day_1km
2022-01-08	33.39
2022-01-09	30.47
2022-01-10	30.47
2022-01-11	30.47
2022-01-12	30.47
...	...
2022-12-23	27.87
2022-12-24	33.23
2022-12-25	30.47
2022-12-26	30.47
2022-12-27	24.61

354 rows × 1 columns

In [ ]:
```
median_data['LST_Day_1km'].plot(color='red', marker='o', linestyle='dotted', fig

sample_data['LST_Day_1km'].plot(ylabel='LST_Day_1km', marker='o', color='blue',
plt.xlabel('DateTimeIndex')
```

Out[ ]:
```
Text(0.5, 0, 'DateTimeIndex')
```



Imputation of continuous statistics

This method replaces missing values with a continuous statistic (such as mean, median, or mode) over a specified window period. Commonly used on time series data, it assumes that data points closer in time are more similar. While this method can handle non-random confounding and preserve temporal dependence, the choice of window size and statistic can significantly affect the results, making it crucial to select these parameters carefully. This method may not be effective for data with large gaps in missing values.

In [ ]:
```
rolling_mean_data = sample_data.fillna(sample_data.rolling(7, min_periods=1).mea
```

In [ ]:
```
rolling_mean_data
```

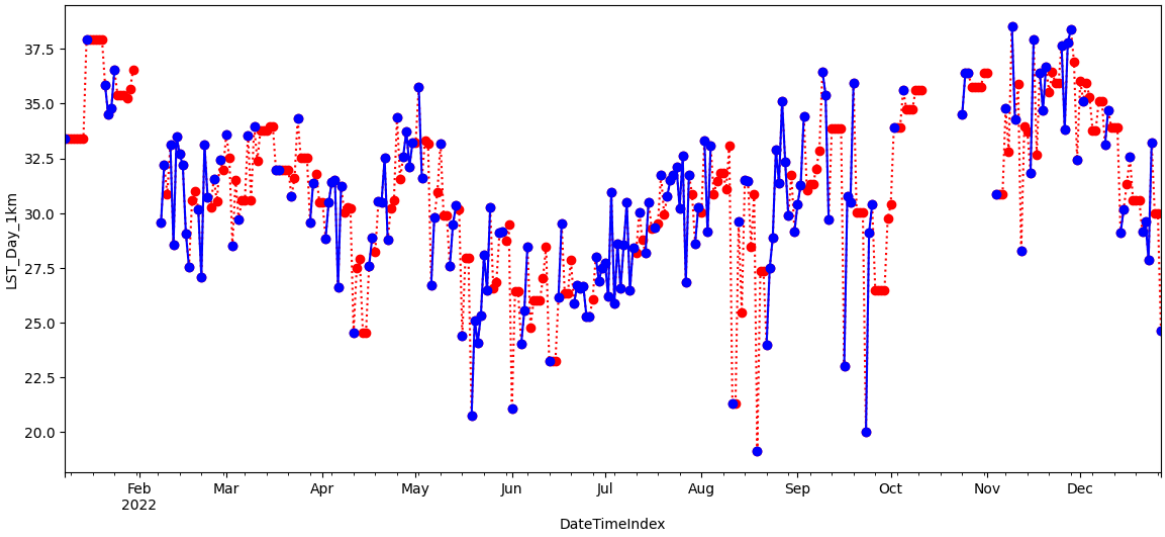
Out[ ]:

datetime	LST_Day_1km
2022-01-08	33.390
2022-01-09	33.390
2022-01-10	33.390
2022-01-11	33.390
2022-01-12	33.390
...	...
2022-12-23	27.870
2022-12-24	33.230
2022-12-25	29.965
2022-12-26	29.965
2022-12-27	24.610

354 rows × 1 columns

In [ ]:
```
rolling_mean_data['LST_Day_1km'].plot(color='red', marker='o', linestyle='dotted

sample_data['LST_Day_1km'].plot(ylabel='LST_Day_1km', marker='o', color='blue',
plt.xlabel('DateTimeIndex')
```

Out[ ]:
```
Text(0.5, 0, 'DateTimeIndex')
```



Nearest Neighbor

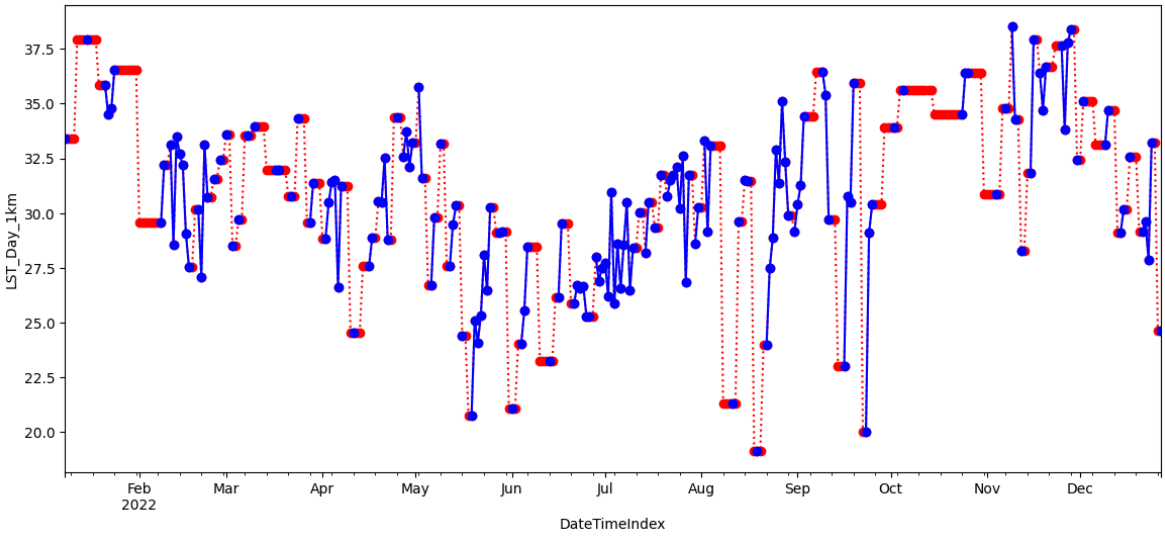
Nearest interpolation is a method that estimates missing values by taking the value of the nearest neighboring data point. This approach can be valuable when data exhibit abrupt changes or irregular intervals. This method preserves the original values and details of the data, but it also creates sharp discontinuities, which can affect the accuracy and quality of the interpolated curve.

Generally, KNN is a versatile algorithm used for supervised ML tasks. An effective approach to data imputation is to use a model to predict missing values. Specifically, a new sample is imputed by finding the samples in the training set that are “closest” to it and averaging these nearby points to fill in the value. However, KNN imputation also has some limitations, such as being sensitive to outliers, noise, and scale, requiring a large and/or representative sample size.

In [ ]:
```
nearest_data = sample_data.interpolate(method='nearest')
```

In [ ]:
```
nearest_data['LST_Day_1km'].plot(color='red', marker='o', linestyle='dotted', fi

sample_data['LST_Day_1km'].plot(ylabel='LST_Day_1km', marker='o', color='blue',
plt.xlabel('DateTimeIndex')
```

Out[ ]:
```
Text(0.5, 0, 'DateTimeIndex')
```



Here we will use the KNN Imputer algorithm:

In [ ]:
```
from sklearn.impute import KNNImputer
```

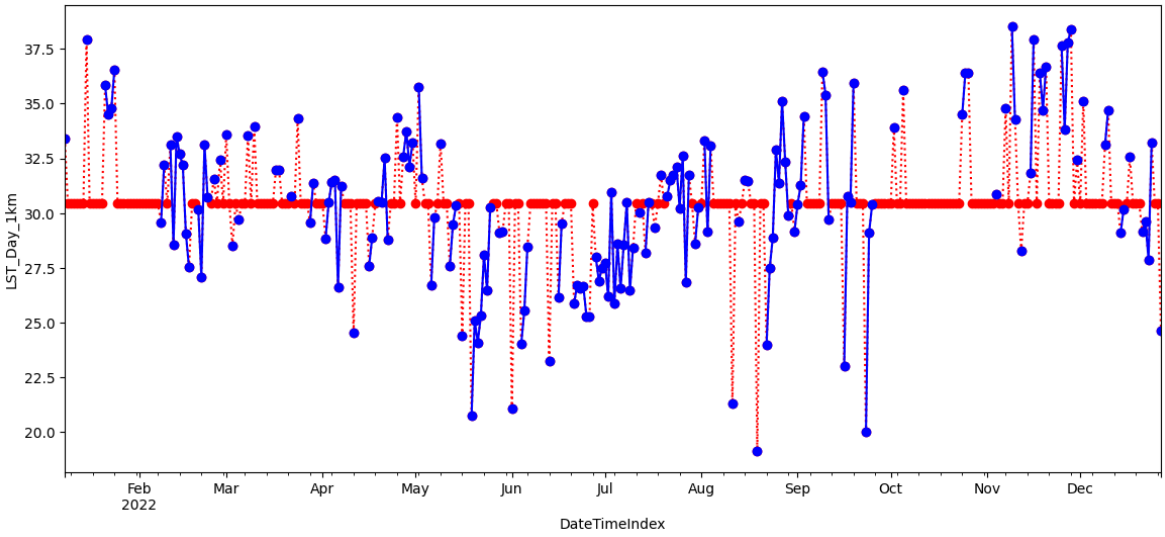
In [ ]:
```
knn_imputer = KNNImputer(n_neighbors=3)
```

In [ ]:
```
knn_data = sample_data.copy()
```

In [ ]:
```
knn_data[['LST_Day_1km']] = knn_imputer.fit_transform(knn_data[['LST_Day_1km']])
```

In [ ]:
```
knn_data['LST_Day_1km'].plot(color='red', marker='o', linestyle='dotted', figsiz

sample_data['LST_Day_1km'].plot(ylabel='LST_Day_1km', marker='o', color='blue',
plt.xlabel('DateTimeIndex')
```

Out[ ]:
```
Text(0.5, 0, 'DateTimeIndex')
```



Time Series Interpolation

Interpolation is a technique commonly used in time series analysis to estimate missing values between two known data points. Time series data often contains gaps, either due

to missing data or irregular sampling intervals. Interpolation helps fill these gaps by estimating the value of a missing data point based on the values of adjacent data points.

There are several interpolation methods that can be used in time series analysis, including linear interpolation, polynomial interpolation, and spline interpolation.

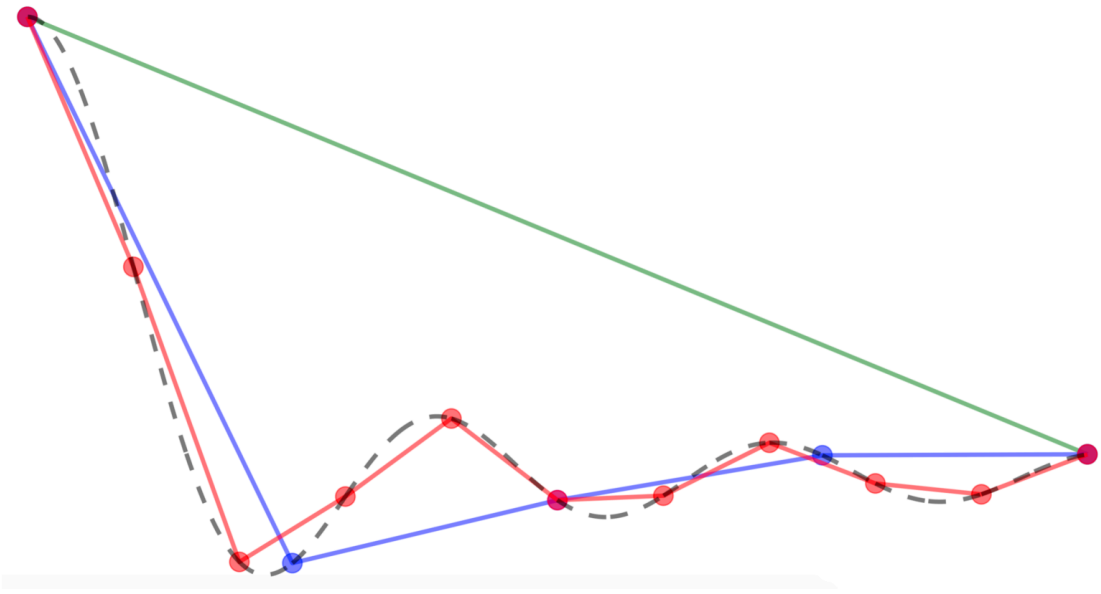
Interpolation can be a useful tool for analyzing time series data because it allows for the creation of a complete, continuous time series. However, it is important to exercise caution when using interpolation because it can introduce errors if the underlying assumptions of the interpolation method are not met. It is also essential to consider the potential impact of missing data on the analysis and interpretation of results.

In summary, interpolation is a valuable technique for filling in missing values in time series data. However, it is important to use appropriate methods and consider the potential limitations and impact of missing data on the analysis.

Interpolation Types

Linear interpolation

Linear interpolation is the most straightforward and commonly used interpolation method. It comes naturally to us when we have two points, we connect them with a straight line to fill in the missing information between them. In doing so, we have made our assumption that the points on the line represent the unobserved values. When there are more than two points, we simply connect any pair of adjacent points with straight lines. This is piecewise interpolation in the sense that in each subinterval formed by two adjacent points, we use a different line segment to represent the missing values. It is possible for all the points to lie on the same line, but we still go through the procedure of connecting each individual pair of points.



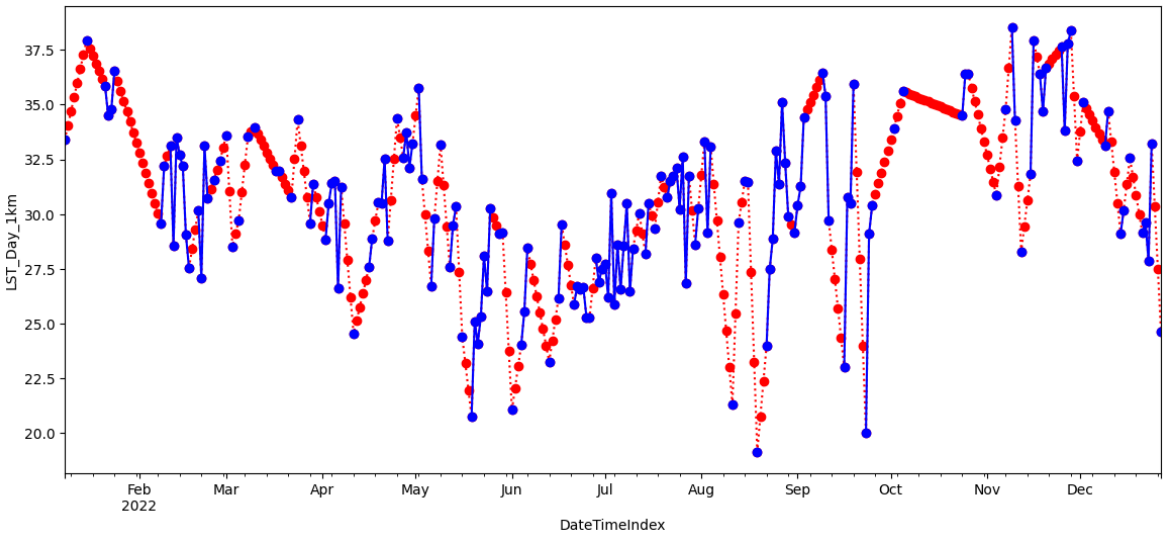
Linear interpolation works best when we have a lot of points. When there are more points, a drastic change in the values of two adjacent points is less likely. Some discrepancies between any two adjacent points are small and can therefore be ignored. This is also why linear interpolation is the default method for visualizing discrete data points.

When we have fewer points, the search for greater accuracy prevails. And the errors between the interpolated values and their true values have a greater impact on the analysis. Thus, more interpolation methods are created to meet such demands.

In [ ]:
```
linear_data = sample_data.interpolate(method='linear').copy()
```

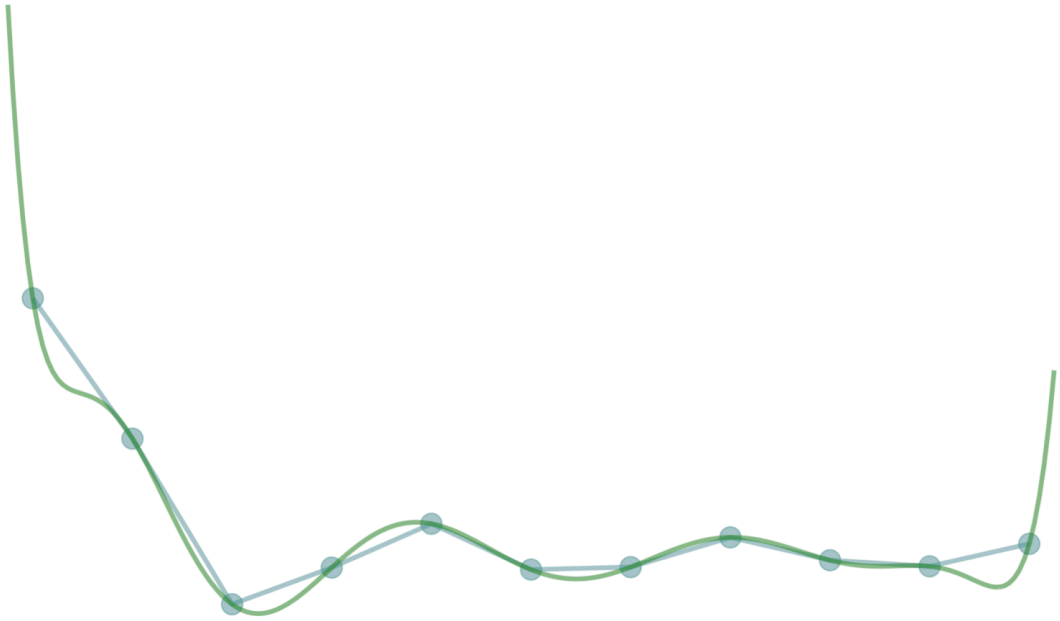
In [ ]:
```
linear_data['LST_Day_1km'].plot(color='red', marker='o', linestyle='dotted', fig

sample_data['LST_Day_1km'].plot(ylabel='LST_Day_1km', marker='o', color='blue',
plt.xlabel('DateTimeIndex')
```

Out[ ]:
```
Text(0.5, 0, 'DateTimeIndex')
```



Polynomial interpolation

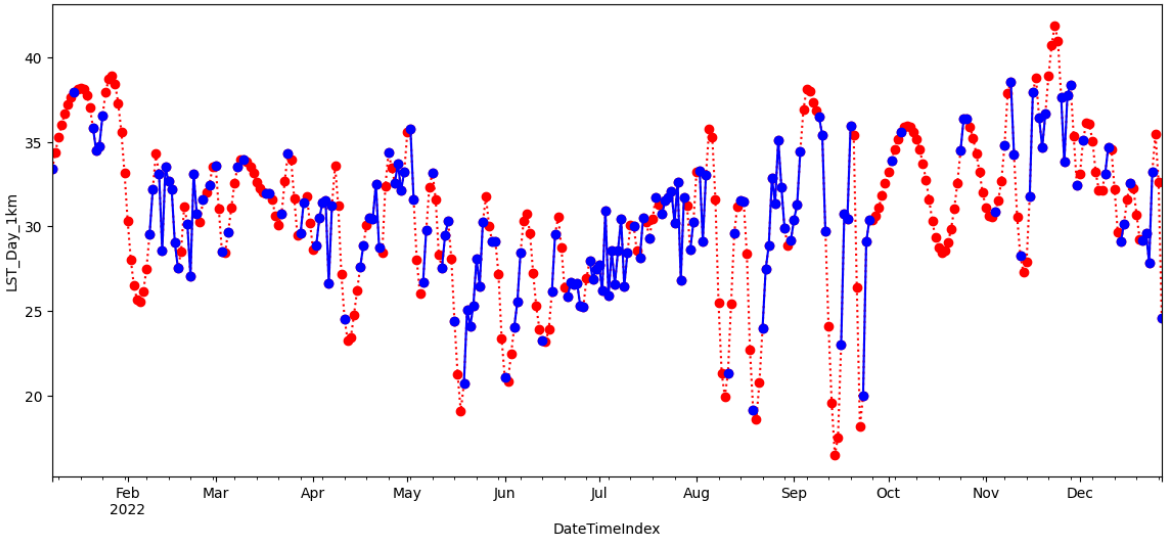
Polynomial interpolation assumes that the points are samples taken from a polynomial curve. We know that a straight line can be described by a linear function, which is a special case of a polynomial of degree 1, and that a parabola can be described by a quadratic function, which is a polynomial of degree 2. We also know that any two points determine a line. Meanwhile, we can describe three or more points by a linear function if all of these points lie on the same line. Therefore, N points can be described by a polynomial of at most N-1 degrees. Consequently, there must exist a polynomial with the smallest number of degrees that describes all the points. Such a polynomial, given the observed points, is called the Lagrange polynomial.



In [ ]:
```
quadratic_data = sample_data.interpolate(method='quadratic').copy()
```

In [ ]:
```
quadratic_data['LST_Day_1km'].plot(color='red', marker='o', linestyle='dotted',

sample_data['LST_Day_1km'].plot(ylabel='LST_Day_1km', marker='o', color='blue',
plt.xlabel('DateTimeIndex')
```

Out[ ]:
```
Text(0.5, 0, 'DateTimeIndex')
```



Spline Interpolation

In this method, missing values are replaced based on a spline interpolation of the available values. Spline interpolation employs piecewise polynomials to approximate the data, capturing nonlinear patterns. This method is suitable for time series data, but assumes some smoothness in the data.

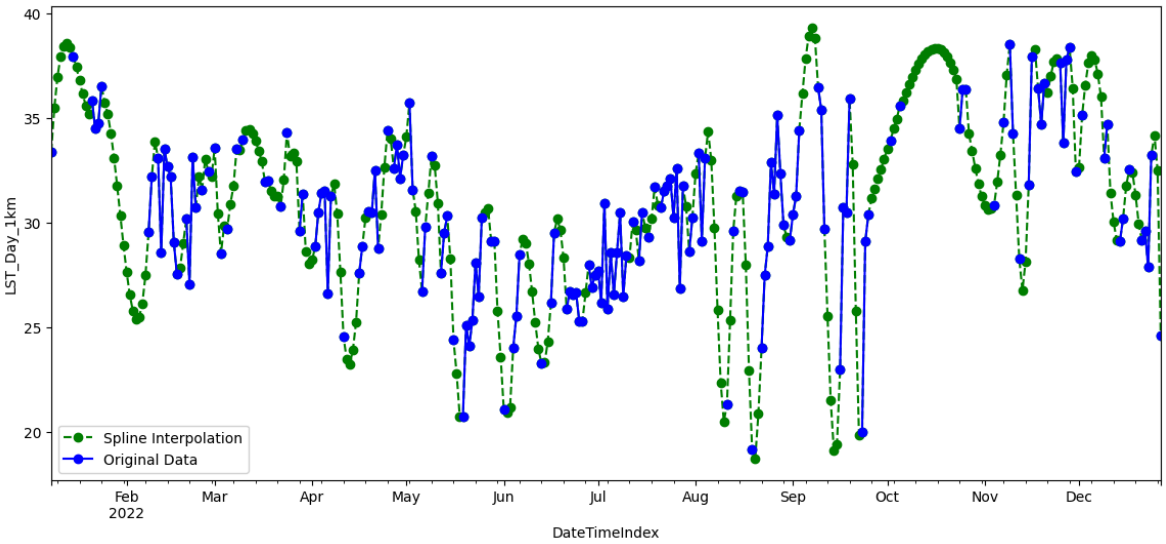
Spline interpolation is a method in which data is fitted using splines, which are piecewise polynomial functions, with the goal of creating a smooth curve that passes through all of the given data points. In other words, instead of using a single polynomial to interpolate

all of the points, as in polynomial interpolation, spline interpolation uses different low-degree polynomials for each interval between the points, joining them smoothly at the data points. This allows the curve to be more flexible and better capture the variations in the data, avoiding unwanted oscillations.

- Data division: The data range is divided into subranges, with known data points serving as nodes.
- Spline fitting: A spline, which is a low-degree polynomial, usually cubic, is fitted to each subrange.
- Smoothness conditions: Smoothness conditions are imposed on the nodes to ensure that the spline curve is continuous and smooth across the data range. These conditions usually involve matching the derivatives of the splines at the nodes.
- Interpolation: Once the splines are fitted and the smoothness conditions are applied, the resulting spline curve can be used to interpolate values at any point within the data range.

In [ ]:
```
spline_data = sample_data.interpolate(method='spline', order=3).copy()
```

In [ ]:
```
spline_data['LST_Day_1km'].plot(color='green', marker='o', linestyle='dashed', f

sample_data['LST_Day_1km'].plot(ylabel='LST_Day_1km', marker='o', color='blue',
plt.xlabel('DateTimeIndex')
plt.legend(['Spline Interpolation', 'Original Data'])
plt.show()
```



Seasonal Trend Decomposition Using Loess Imputation (STL)

Seasonal Trend Decomposition Using Loess (STL) is a statistical method for decomposing a time series into three components: trend, seasonal, and remainder (random). It can be used to impute missing data in a time series. In the STL imputation method, missing values are first estimated using interpolation to allow for STL decomposition. Then, the seasonal and trend components of the decomposed time series are extracted. The

missing values are then re-estimated by interpolating the trend component and re-adding the seasonal component.

STL imputation can be used when dealing with time series data that exhibit a seasonal pattern. It is particularly useful when the data is missing at random and the missingness is unrelated to the trend or seasonality of the time series.

In [ ]:
```
from statsmodels.tsa.seasonal import STL
```

In [ ]:
```
stl_data = sample_data.copy()
```

We fill missing values in the time series

In [ ]:
```
imputed_indices = stl_data[stl_data['LST_Day_1km'].isnull()].index
```

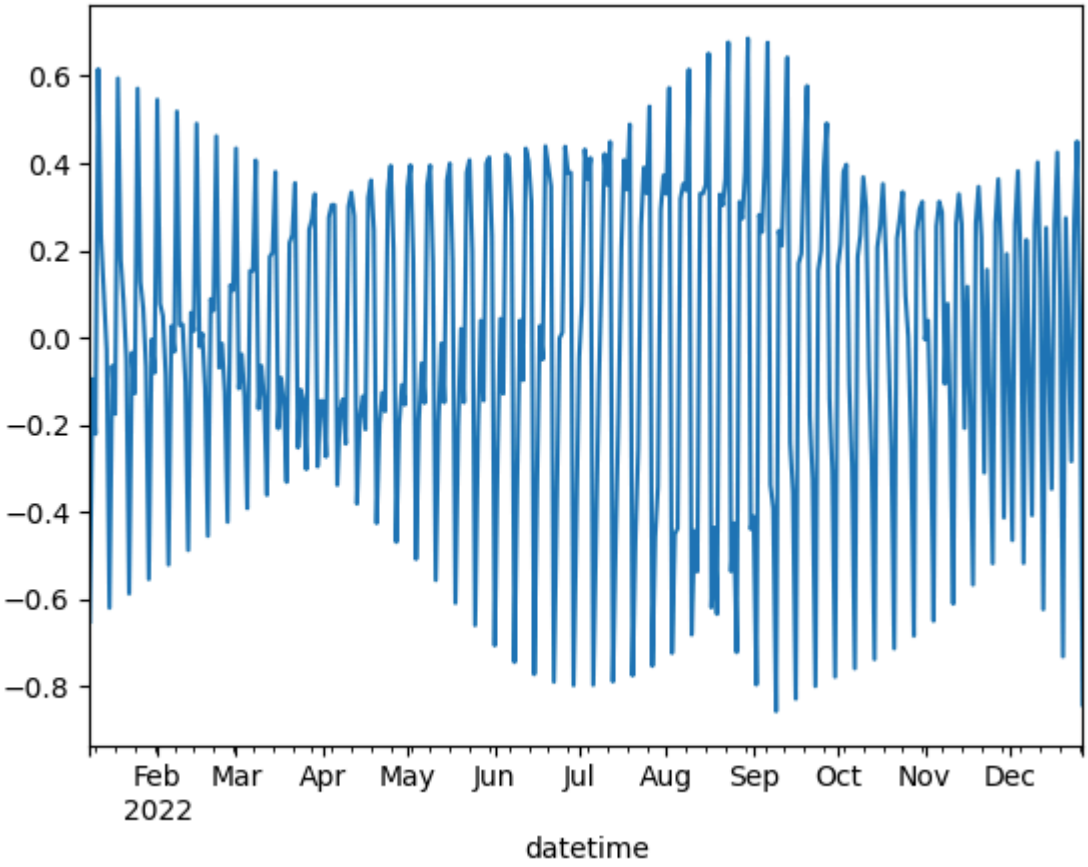
Extract seasonal and trend components

In [ ]:
```
stl = STL(sample_data['LST_Day_1km'].interpolate(), seasonal=31)
res = stl.fit()
```

In [ ]:
```
seasonal_component = res.seasonal
```

In [ ]:
```
seasonal_component.plot()
```

Out[ ]:
```
<Axes: xlabel='datetime'>
```



Create the seasonally adjusted series

In [ ]:
```
df_deseasonalised = stl_data['LST_Day_1km'] - seasonal_component
```

Interpolate missing values in the seasonally adjusted series

In [ ]:
```
df_deseasonalised_imputed = df_deseasonalised.interpolate(method="linear")
```

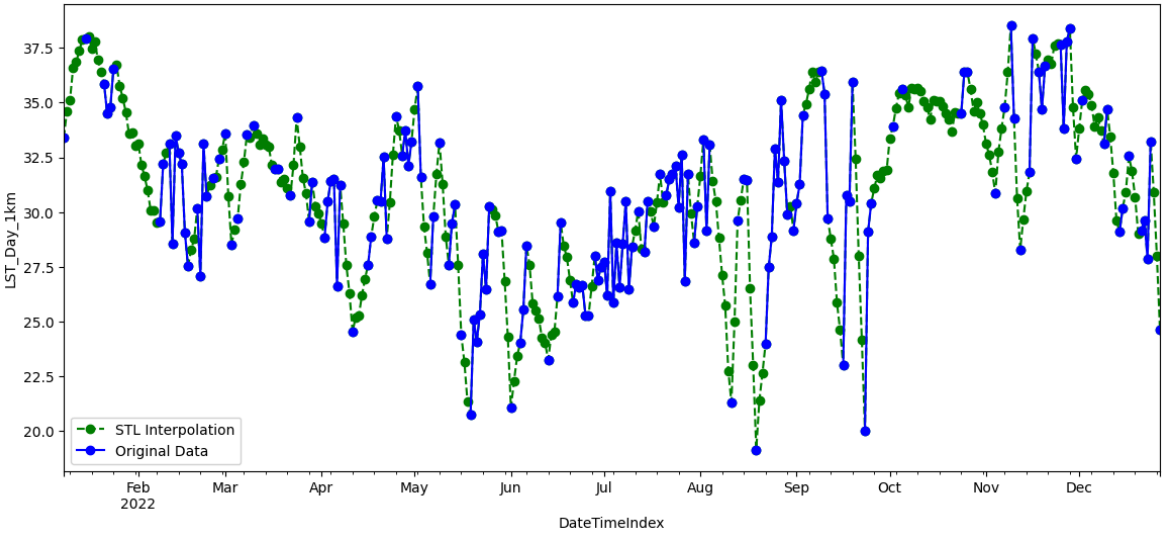
Add the seasonal component back in to create the final imputed series

In [ ]:
```
df_imputed = df_deseasonalised_imputed + seasonal_component
```

Update the original dataframe with the imputed values

In [ ]:
```
stl_data.loc[imputed_indices, 'LST_Day_1km'] = df_imputed[imputed_indices]
```

In [ ]:
```
stl_data['LST_Day_1km'].plot(color='green', marker='o', linestyle='dashed', figs

sample_data['LST_Day_1km'].plot(ylabel='LST_Day_1km', marker='o', color='blue',
plt.xlabel('DateTimeIndex')
plt.legend(['STL Interpolation', 'Original Data'])
plt.show()
```



Thank you! See you in the next Chapter!

References: https://www.analyticsvidhya.com/blog/2021/06/power-of-interpolation-in-python-to-fill-missing-values/#h-types-of-interpolation

https://newdigitals.org/2024/03/14/time-series-imputation-interpolation-anomaly-detection/

https://www.d3view.com/interpolation-methods-for-time-series-data/