Data Imputation Demystified | Time Series Data

Ahmed Abulkhair · Follow

21 min read · Jun 13, 2023

Listen Share

Missing values are a common issue in data analysis and can significantly affect the results of your data analysis. They occur when no data value is stored for a variable in an observation. Handling missing values is important as many machine learning algorithms do not support data with missing values.

Types of Missing Values

There are three types of missing data;

1. **MCAR (Missing Completely at Random):** This is a case when the probability of missing values in a variable is the same for all samples. For example, respondents of data collection process decide whether to include their house number in a survey completely at random. Here, all missing values are resulted from random chance. It is a rare situation.

2. **MAR (Missing at Random):** This is a case when variable has missing values that are randomly distributed, however, they are related to some other variables' values. For example, if men are more likely to hide their income level, income is MAR. The missingness of income data depends on the gender of the respondent.

3. **MNAR (Missing not at Random):** This is a case when the missingness of data is related to events or factors which are not measured by the researcher. For example, people with high levels of income are less likely to respond to income level question due to the fear of higher taxes. Here, the missingness of income data is not at random, it is related to the income level, but income level is not known to us.

While missing values often have a negative connotation in machine learning and analysis, they're not always detrimental. In fact, through appropriate management, these missing values can transform into valuable features, yielding additional insights and enhancing the predictive capacity of machine learning models. The skill of addressing missing data typically relies on two key pieces of information: the reason behind the missing values and the most effective method to manage them.

Time Series Data

Time series data is a sequence of data points in chronological order that is used to analyze trends and patterns over time. This type of data is different from cross-sectional data which is data collected at a single point in time and panel data which is a combination of time series and cross-sectional data.

Why are Missing Values Special in Time Series Data?

Temporal dependence characterizes time series data because observations close in time tend to be similar compared to cross-sectional data. However, this dependency

gets disrupted by missing values. Time series also display trends and seasonal patterns affected by missing values that make accurate modeling more challenging.

Imputing missing values for this kind of data differs from cross-sectional methods due to temporal-dependence considerations calling for specialized interpolation methods such as those accounting for the temporal structure in use cases involving imputations of missing values in time-series datasets. Poor handling of these values can significantly affect a model's forecast accuracy preventing meaningful interpretations because future-state estimates heavily rely on past states' uniqueness based on trends or seasonality factors indicating why careful attention is necessary when dealing with them.

```
# Import necessary libraries
import pandas as pd
import numpy as np
from datetime import datetime, timedelta
import random
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
from statsmodels.tsa.seasonal import STL
from sklearn.impute import KNNImputer
import seaborn as sns

sns.set_style("darkgrid")
# Set the random seed for reproducibility
np.random.seed(0)

# Define the start date
```

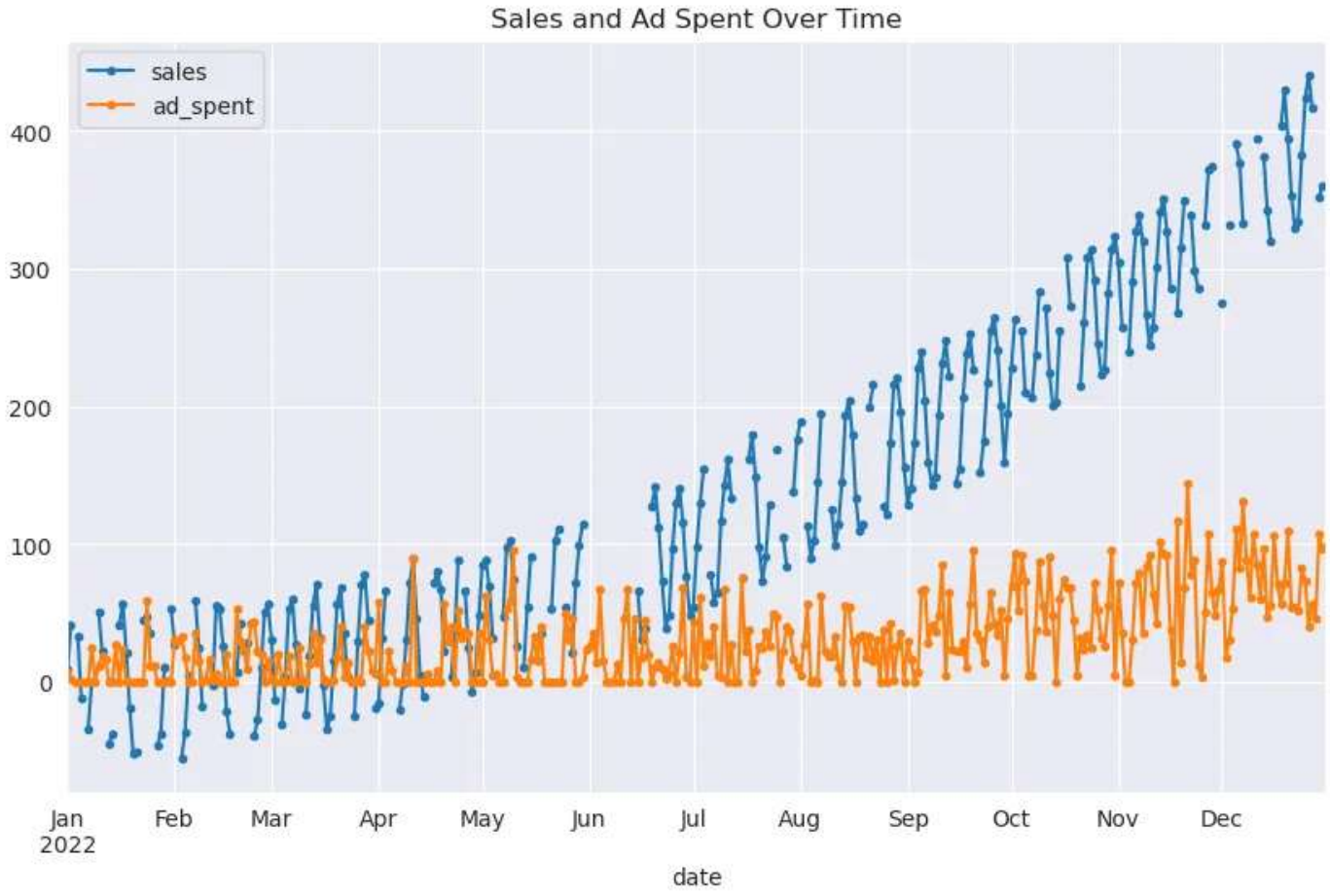


```
# Generate more pronounced trend component (increasing linearly)
trend = np.power(np.linspace(0.1, 20, 365), 2)

# Generate more pronounced seasonal component (sinusoidal pattern)
with weekly period
seasonal = 50 * np.sin(np.linspace(0, 2 * np.pi * 52, 365)) # 52 weeks
in a year

# Generate random noise
noise = np.random.normal(0, 5, 365)

# Combine components to generate sales data
```

```
sales = trend + seasonal + noise

# Create ad_spent feature
# Using a scaled version of sales and adding more noise
ad_spent = 0.2 * sales + np.random.normal(0, 30, 365)  # Increased the
noise and decreased the scale factor
ad_spent = np.maximum(ad_spent, 0)  # Making sure all ad_spent values
are non-negative

# Create a dataframe
df = pd.DataFrame(
    {
        'date': dates,
        'sales': sales,
        'ad_spent': ad_spent
    }
)

# Set the date as the index
df.set_index('date', inplace=True)

# Generate missing values for a larger gap
for i in range(150, 165):  # A 15-day gap
    df.iloc[i, df.columns.get_loc('sales')] = np.nan

# Randomly choose indices for missing values (not including the
already missing gap)
random_indices = random.sample(list(set(range(365)) -
set(range(150,165))), int(0.20 * 365))

# Add random missing values
for i in random_indices:
    df.iloc[i, df.columns.get_loc('sales')] = np.nan

# Display the dataframe
display(df.head())

# Print the percentage of missing values
print('% missing data in sales: ',
100*df['sales'].isnull().sum()/len(df))

# Plot the data
df[['sales', 'ad_spent']].plot(style='.-', figsize=(10,6),
title='Sales and Ad Spent Over Time')
plt.show()

# Print correlation between sales and ad_spent
print("Correlation between sales and ad_spent: ",
df['sales'].corr(df['ad_spent']))
```

date	sales	ad_spent
2022-01-01	8.830262	6.593897
2022-01-02	41.116283	2.503652
2022-01-03	NaN	0.000000
2022-01-04	32.968355	0.000000
2022-01-05	-12.254839	0.000000

```
% missing data in sales:  24.10958904109589
```




```
Correlation between sales and ad_spent:  0.6556099955159692
```

Our dataset comprises daily data spanning one year, featuring two variables: sales and ad_spent. Approximately 25% of the sales data contains missing values, which appear

randomly and, in some cases, constitute complete gaps in the data. On the other hand, the ad_spent feature is devoid of any missing values. A notable correlation of about 66% exists between ad_spent and sales.

Approaches to Handling Missing Values

Various strategies exist to manage missing values, with the most suitable one often dependent on the nature of both the data and the missing values themselves. We'll delve into the following methods:

- **Deletion**: This strategy entails eliminating any rows that contain missing values. Although straightforward to execute, if the missing data isn't Missing Completely at Random (MCAR), this approach may result in valuable information loss.
- **Constant Imputation**: This technique substitutes all missing values with a constant, which might be a common value like zero or an unusual one that effectively establishes a new category for missing values. Although it's simple to apply, it can skew the data's distribution and possibly yield biased estimates. Unless there's a compelling reason to select a particular constant, this method is usually not recommended.
- **Last Observation Carried Forward (LOCF)** and **Next Observation Carried Backward (NOCB)**: These methods replace missing values either with the immediately preceding observed value (LOCF) or the subsequent observed value (NOCB). These are potentially useful for time series data, but they may introduce bias if the data isn't stationary.
- **Mean/Median/Mode Imputation**: In this approach, missing values are replaced with the mean (for continuous data), median (for ordinal data), or mode (for categorical data) of the available values. While easy to implement, this method could potentially underestimate variance.
- **Rolling Statistics Imputation**: This method substitutes missing values with a rolling statistic (like mean, median, or mode) over a specified window period. Commonly used in time series data, it assumes that the data points closest in time are more similar. Although this method can handle non-random missingness 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 of missing values.

- **Linear Interpolation**: Here, missing values are replaced based on a linear equation derived from the available values. This method is appropriate for time series data but presupposes a linear relationship between observations.
- **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 non-linear patterns. This method is suitable for time series data but assumes a certain smoothness in the data.
- **K-Nearest Neighbors (KNN) Imputation**: This technique substitutes missing values based on the values of the "k" nearest neighbors. While it can capture complex patterns in the data, it may be computationally expensive for large datasets.
- **STL Decomposition for Time Series**: This method breaks down the time series into trend, seasonality, and residuals, then imputes missing values in the residuals before reassembling the components. This could prove useful for time series data with a distinct trend and seasonality.

Deletion

Deletion, also known as listwise deletion, is the process of removing observations where at least one variable is missing. This method is the easiest way to handle missing data but it is not generally advised as it can lead to loss of useful information and potentially biased results if the missing data is not MCAR.

When to use it:

Deletion is appropriate when the amount of missing data is very small, such that it represents a negligible portion of the total dataset. It is also used when the missing data is Missing Completely at Random (MCAR), meaning that there are no patterns in the missing data.

Pros:

- Easy to implement.
- Does not introduce bias in the data.

Cons:

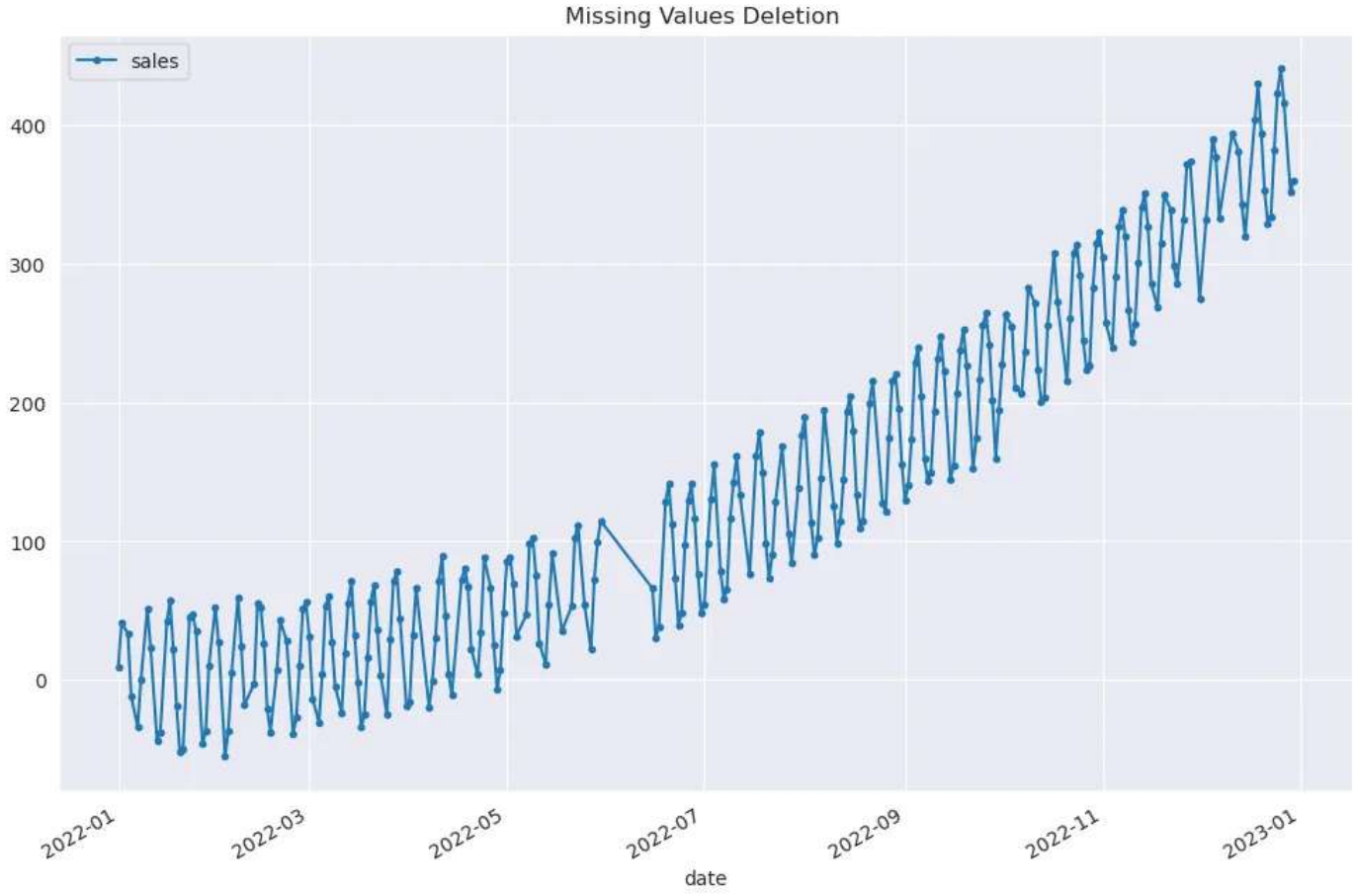
- Can lead to loss of information if the dataset is not large enough.
- Can lead to biased results if the missing data is not MCAR.
- Not suitable for large amounts of missing data.

Effect on model-building process:

Deletion reduces the size of the dataset, which can lead to loss of information and potentially less accurate models. If the missing data is not MCAR, deletion can also introduce bias in the models. Let's implement deletion on our synthetic data.

```
# Create a copy of the dataframe
df_deletion = df.copy()

# Remove rows with missing values
df_deletion.dropna(inplace=True)

# Display the dataframe
display(df_deletion.head())
# plot the data

df_deletion[['sales']].plot(style='.-', figsize=(12, 8),
title='Missing Values Deletion ')
plt.show()
```

date	sales	ad_spent
2022-01-01	8.830262	6.593897
2022-01-02	41.116283	2.503652
2022-01-04	32.968355	0.000000
2022-01-05	-12.254839	0.000000
2022-01-07	-34.157929	0.000000



For time series data, the deletion method is not typically favored for addressing missing values. The substantial alteration it causes to the time-dependent structure of the data could inadvertently eliminate its essential temporal attributes.

Conversely, in cross-sectional data, if the percentage of missing values is relatively low, initiating its handling can provide a clearer understanding of the feature with the missing data.

Constant Imputation

Constant imputation is a method of handling missing data by replacing the missing values with a constant value. Instead of removing the observations with missing data or trying to predict the missing values, constant imputation assigns the same fixed value to all missing entries.

When to use it:

Constant imputation is typically used when the missing data has a specific meaning or when the missingness is believed to be informative. For example, if a survey asks respondents about their income and some respondents choose not to disclose their

income, the missing values can be replaced with a constant value such as “-1” to indicate that the income was not disclosed.

Pros:

- Easy to implement.
- Preserves the structure and size of the dataset.
- Can capture the meaning or significance of missing values in certain cases.

Cons:

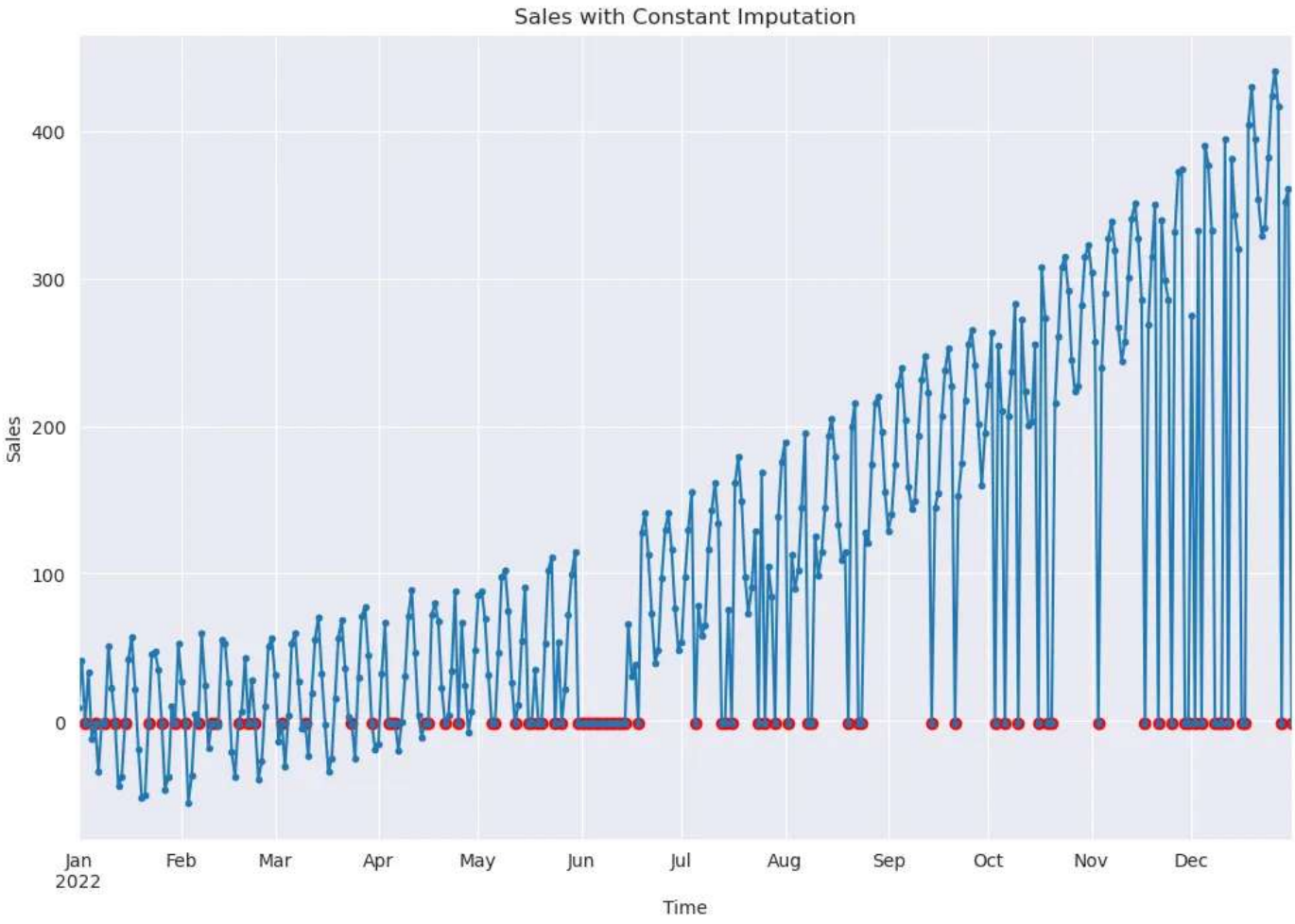
- Introduces bias in the data if the constant value is not truly representative of the missing values.
- Can distort statistical analysis and modeling results.
- Does not provide any additional information about the missing values.

Impact on the Model-Building Process:

Constant imputation can significantly impact statistical analysis and model development by introducing bias and altering the relationships between variables. In this approach, missing values are essentially assumed to be known and equivalent to a constant value. This could lead to biased estimates and incorrect model predictions if the selected constant value does not accurately reflect the actual missing values. Furthermore, it’s worth noting that Linear Models tend to be more adversely impacted by constant imputation compared to Tree-Based Models. The reason being, Tree-Based Models inherently aim to partition the examples, and having some examples with the same constant value might enhance the model’s performance.

```
# Apply the forward fill method
df_imputed = df.fillna(-1)

# Plot the main line with markers
df_imputed['sales'].plot(style='.-', figsize=(12,8), title='Sales with
Constant Imputation')

# Add points where data was imputed with red color
plt.scatter(df_imputed[df['sales'].isnull()].index,
df_imputed[df['sales'].isnull()]['sales'], color='red')

# Set labels
plt.xlabel('Time')
```

```
plt.ylabel('Sales')

plt.show()
```



As you can see here, the `constant imputation` has hugly distorted the temporal dependence which will be a serious problem when fitting a statistical model.

Last Observation Carried Forward (LOCF)

LOCF is a method where missing values are replaced with the last observed value. It's a type of imputation that can be used when the data has a temporal or sequential order, and it's reasonable to assume that the missing value would be similar to the previous value.

When should we use it?

LOCF can be useful in time-series data where there is a trend or pattern that continues from one time point to the next. It's also useful when the data is collected at regular intervals, and the values are relatively stable or change slowly over time.

Pros

- Simple and easy to implement.
- Does not require any model fitting.
- Can be a reasonable assumption in certain time-series datasets.

Cons

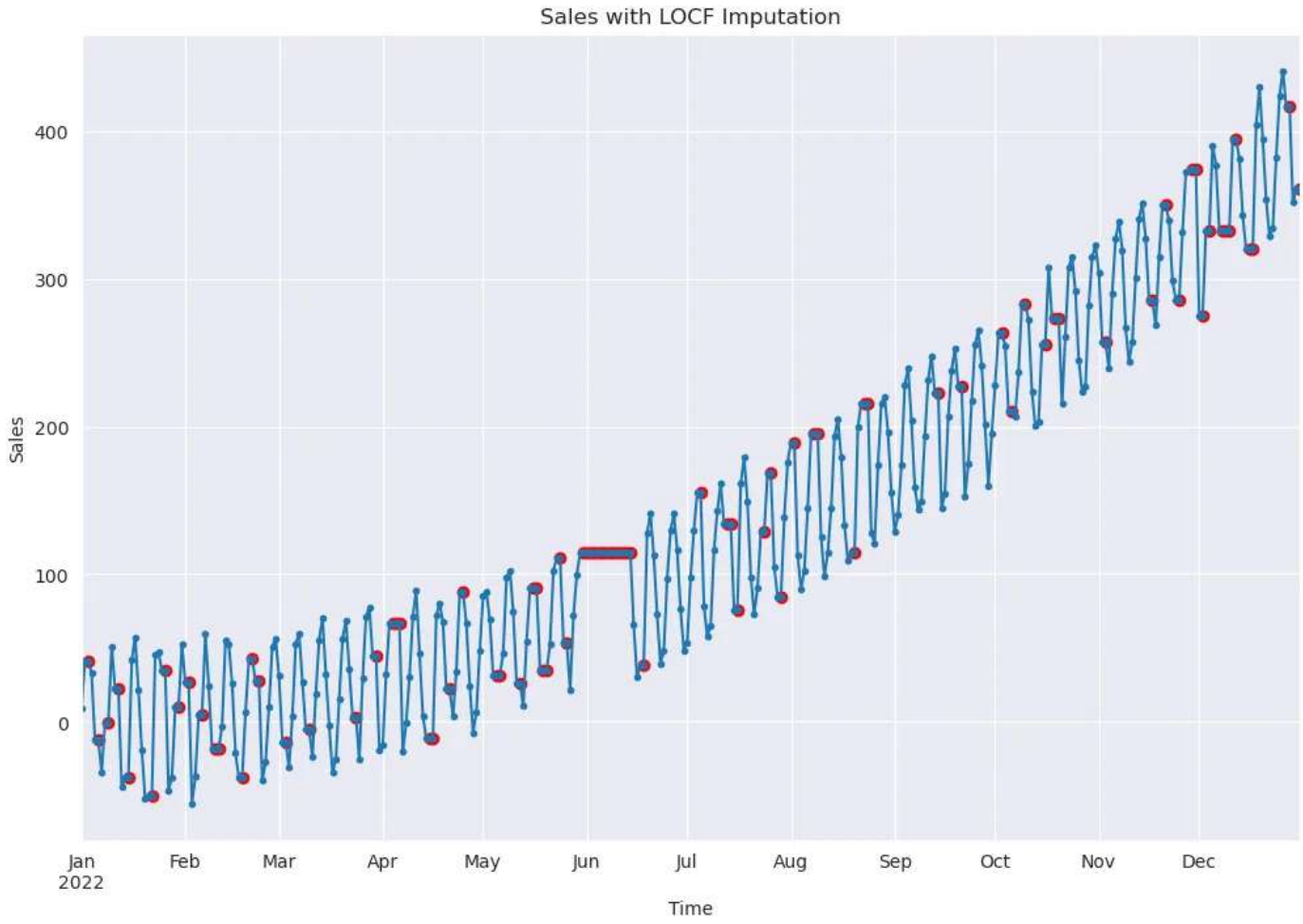
- It can introduce bias into the data if the assumption of similar adjacent values does not hold.
- It does not consider the possible variability around the missing value.
- It can lead to overestimation or underestimation of the data analysis results if the missing data is not random.

Impact on the Model-Building Process:

Using LOCF can affect the model-building process by potentially introducing bias. If the missing data is not random, this method can lead to overestimation or underestimation of the model parameters. It's important to check the assumptions of this method before using it in model building.

Let's see a demo using code.

```
# Apply the forward fill method
df_imputed = df.fillna(method="ffill")

# Plot the main line with markers
df_imputed['sales'].plot(style='.-', figsize=(12,8), title='Sales with
LOCF Imputation')

# Add points where data was imputed with red color
plt.scatter(df_imputed[df['sales'].isnull()].index,
df_imputed[df['sales'].isnull()]['sales'], color='red')

# Set labels
plt.xlabel('Time')
plt.ylabel('Sales')

plt.show()
```



Next Observation Carried Backward (NOCB)

Next Observation Carried Backward (NOCB) is another simple method for handling missing data. This method is the opposite of LOCF. Instead of filling the missing value with the last observation, it fills it with the next observation. It's like looking into the future and filling the missing value with the value that is going to happen.

When should we use it?

NOCB can be used when the data shows a decreasing trend. If the next value is consistently lower than the previous one, then using the next value might be a better option.

Pros

- Simple and easy to implement.
- Not Computationally expensive.

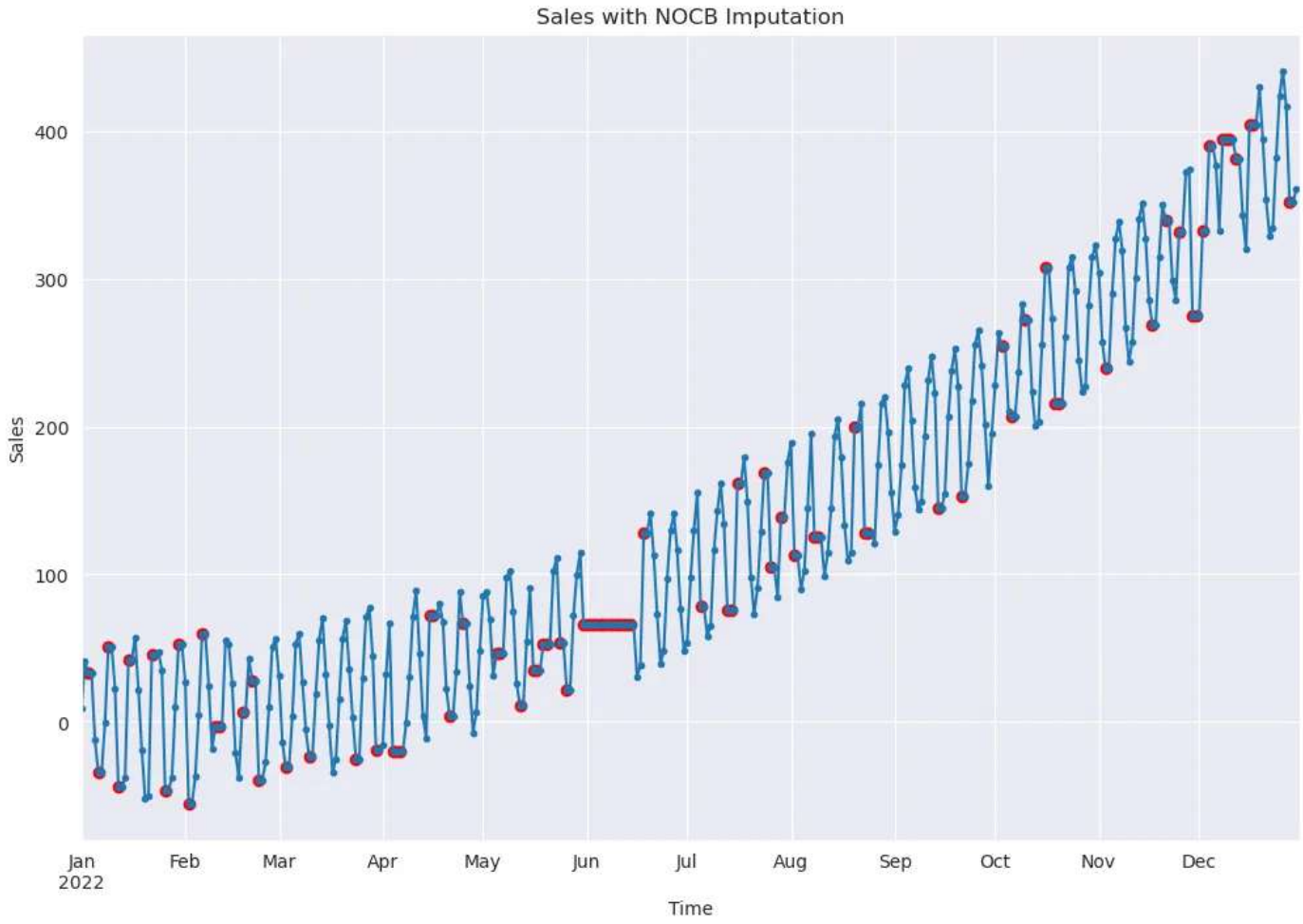
Cons

- Like LOCF, it can lead to bias in the data.

- It’s not suitable for data with high variance.
- It can’t predict the future values, i.e., it can’t fill the missing values if the missing value is the last observation because next value, simply, does not exist in prediction.

Let’s see how NOCB works with our synthetic data.

```
# Apply the backward fill method
df_imputed = df.fillna(method="bfill")

# Plot the main line with markers
df_imputed['sales'].plot(style='.-', figsize=(12,8), title='Sales with
NOCB Imputation')

# Add points where data was imputed with red color
plt.scatter(df_imputed[df['sales'].isnull()].index,
df_imputed[df['sales'].isnull()]['sales'], color='red')

# Set labels
plt.xlabel('Time')
plt.ylabel('Sales')

plt.show()
```



Clearly, LOCF (Last Observation Carried Forward) and NOCB (Next Observation Carried Backward) are plausible choices when dealing with random missing values that have a fairly low likelihood of appearing at these specific points. The distortion they introduce to the dataset is minimal in such instances. However, if missing values form continuous gaps, both methods can cause significant distortion in the data, effectively obliterating any temporal dependencies.

A critical aspect to note is that implementing NOCB in production systems is practically challenging. This is because, during prediction, we won't have the subsequent value available to impute any current missing values.

Mean/Median/Mode Imputation

Mean/Median/Mode imputation is one of the most frequently used methods to handle missing data. In this method, the missing values are replaced with the mean, median, or mode of the variable. Mean imputation is used for continuous variables, while median and mode imputation is used for ordinal and categorical variables, respectively.

When should we use it?

Mean/Median/Mode imputation can be used when the data is missing completely at random (MCAR). If the missing data is not random, this method can lead to biased and inefficient estimates.

Pros

- Easy to implement.
- If the percentage of missing values are not huge then, this will affect the mean, median, and mode of the original data otherwise, it will significantly change the summuary stats of the data.

Cons

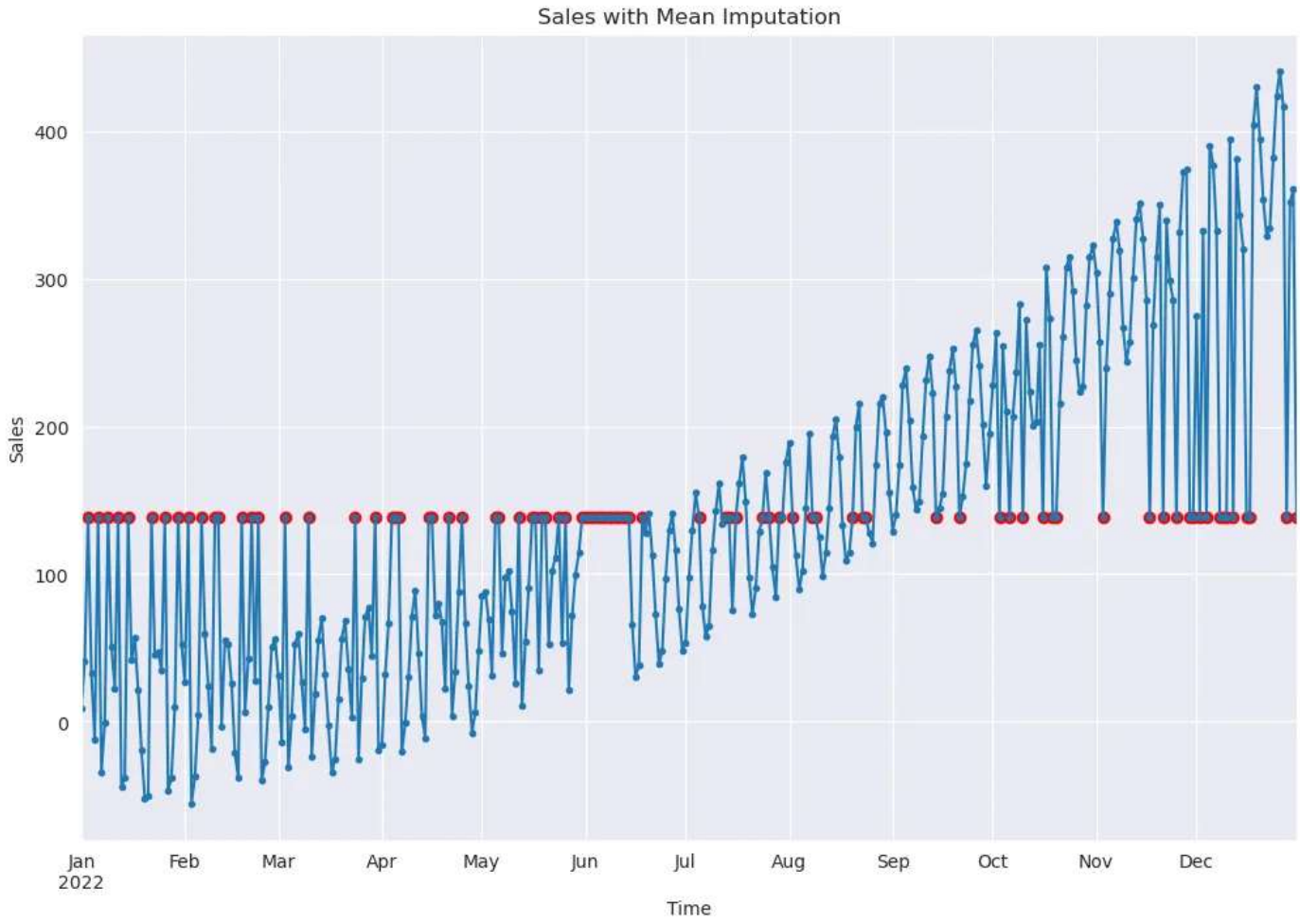
- It can lead to an underestimate of the variance.
- It does not factor the correlations between features.
- It can lead to biased estimates of the covariances and correlations.

Impact on the Model-Building Process:

Mean/Median/Mode imputation can lead to an underestimate of the variance, which can lead to biased estimates of the covariances and correlations. This can affect the model's ability to accurately capture the relationships between variables. Therefore, it's important to carefully consider the nature of the missing data before using this method.

```
# Apply the mean imputation method
df_imputed = df.fillna(df['sales'].mean())

# Plot the main line with markers
df_imputed['sales'].plot(style='.-', figsize=(12,8), title='Sales with
Mean Imputation')

# Add points where data was imputed with red color
imputed_indices = df[df['sales'].isnull()].index
plt.scatter(imputed_indices, df_imputed.loc[imputed_indices, 'sales'],
color='red')

# Set labels
plt.xlabel('Time')
plt.ylabel('Sales')
```

```
plt.show()
```



Clearly, the application of mean imputation significantly distorts the temporal relationships among data points. This is because it fails to account for the time series dependencies inherent in the data, instead, it primarily focuses on the overall mean of the dataset.

Rolling Statistics Imputation

Rolling Statistics Imputation is a method often used in time series data to handle missing data. It leverages the temporal structure of the data by replacing missing values with a rolling statistic (like mean, median, or mode) over a defined window period. Most commonly, a rolling mean (or moving average) is used, which takes the average of the previous ’n’ points.

When should we use it?

Rolling Statistics Imputation is particularly effective for time series data with temporal dependencies. It can handle situations where the missing data is Missing At Random (MAR) or Not Missing At Random (NMAR). This method assumes that nearby points in time are more similar to each other, which often holds true for time series data.

Pros

- Takes into account the temporal structure of the data, preserving the temporal dependence.
- Can handle non-random missingness, making it more versatile than mean/median/mode imputation.
- Is relatively simple to implement with flexible choices for the rolling window and the statistic used.

Cons

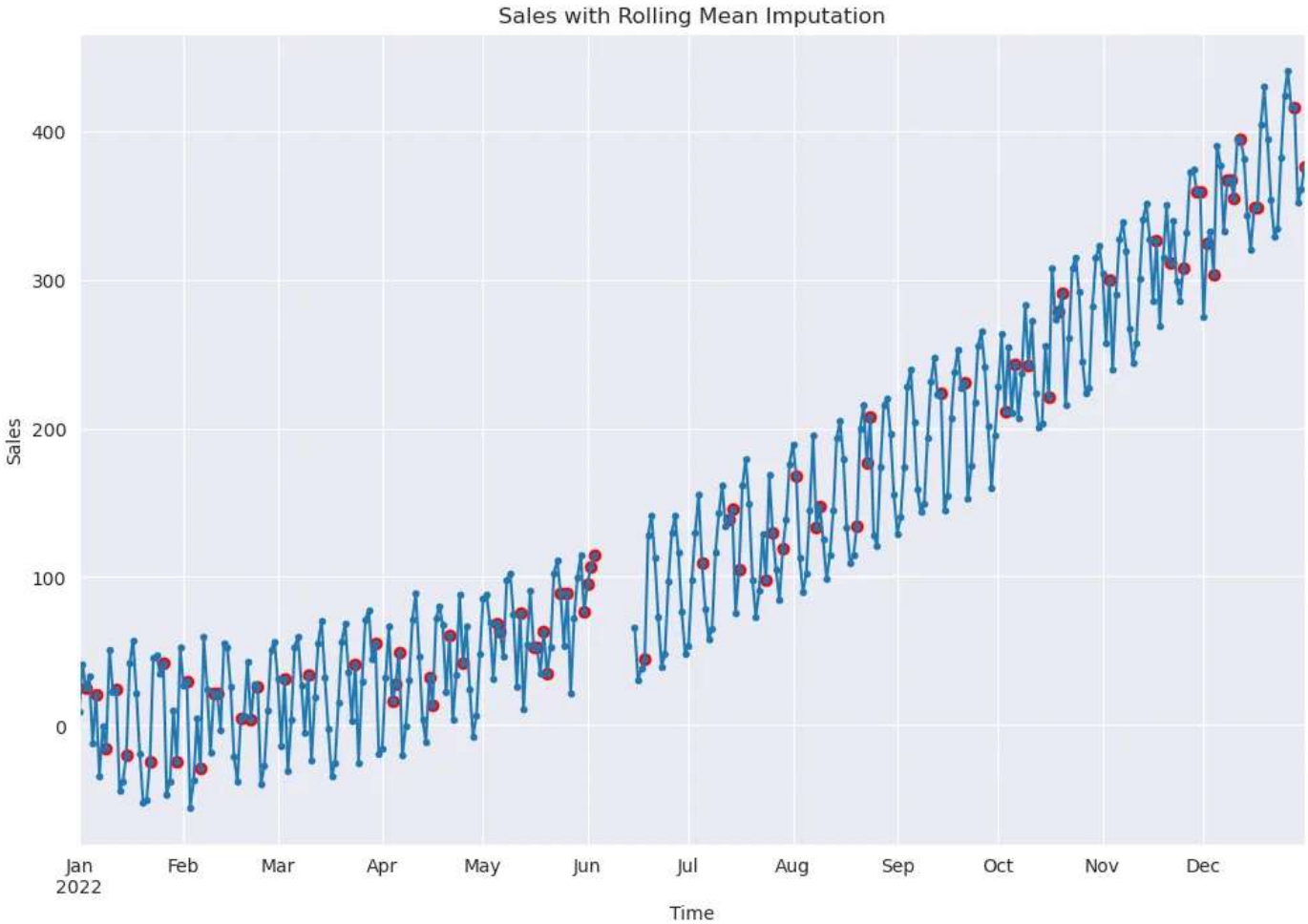
- The choice of window size can significantly impact the results. If the window size is too small, it may fail to capture the underlying trend; if too large, it may smooth out important short-term fluctuations.
- Can still introduce bias if the missing data pattern is complex and not well-captured by the chosen window size and statistic.
- Doesn't work well for data with large gaps of missing values, as the rolling statistic would be imputed with NaN values.

Impact on the Model-Building Process:

Rolling Statistics Imputation can preserve the temporal dependencies in the data, which is beneficial for modeling time series. However, the choice of window size and statistic can significantly affect the results. Therefore, it is important to carefully consider these parameters and validate the imputation method through a rigorous model evaluation process. It is also critical to acknowledge that this method may not be suitable for handling large gaps of missing data.

```
# Make a copy of the original DataFrame
df_copy = df.copy()

# Mark the missing values before imputation
imputed_indices = df_copy[df_copy['sales'].isnull()].index
```

```
# Apply the rolling mean imputation method
df_copy['sales'] =
df_copy['sales'].fillna(df_copy['sales'].rolling(window=4,
min_periods=1).mean().shift(1))

# Plot the main line with markers
df_copy['sales'].plot(style='.-', figsize=(12,8), title='Sales with
Rolling Mean Imputation')

# Add points where data was imputed with red color
plt.scatter(imputed_indices, df_copy.loc[imputed_indices, 'sales'],
color='red')

# Set labels
plt.xlabel('Time')
plt.ylabel('Sales')

plt.show()
```



As you can see, due to the choice of window size, this method is not affective at all in the existance of gaps. Although, a wider choice of the window size may help but it will lead to distortion as well.

Linear Interpolation

Linear interpolation is a method of curve fitting used to estimate the value between two known values. In the context of missing data, linear interpolation can be used to estimate the missing values by drawing a straight line between two points.

When should we use it?

Linear interpolation is suitable when the data shows a linear trend between the observations. It's not suitable for data that shows a nonlinear trend or seasonal pattern as it will almost distort the seasonal pattern at all.

Pros

- Simple and fast.
- It can provide a good estimate for the missing values when the data shows a linear trend.

Cons

- It can provide poor estimates when the data does not show a linear trend.
- It's not suitable for data with a seasonal pattern.

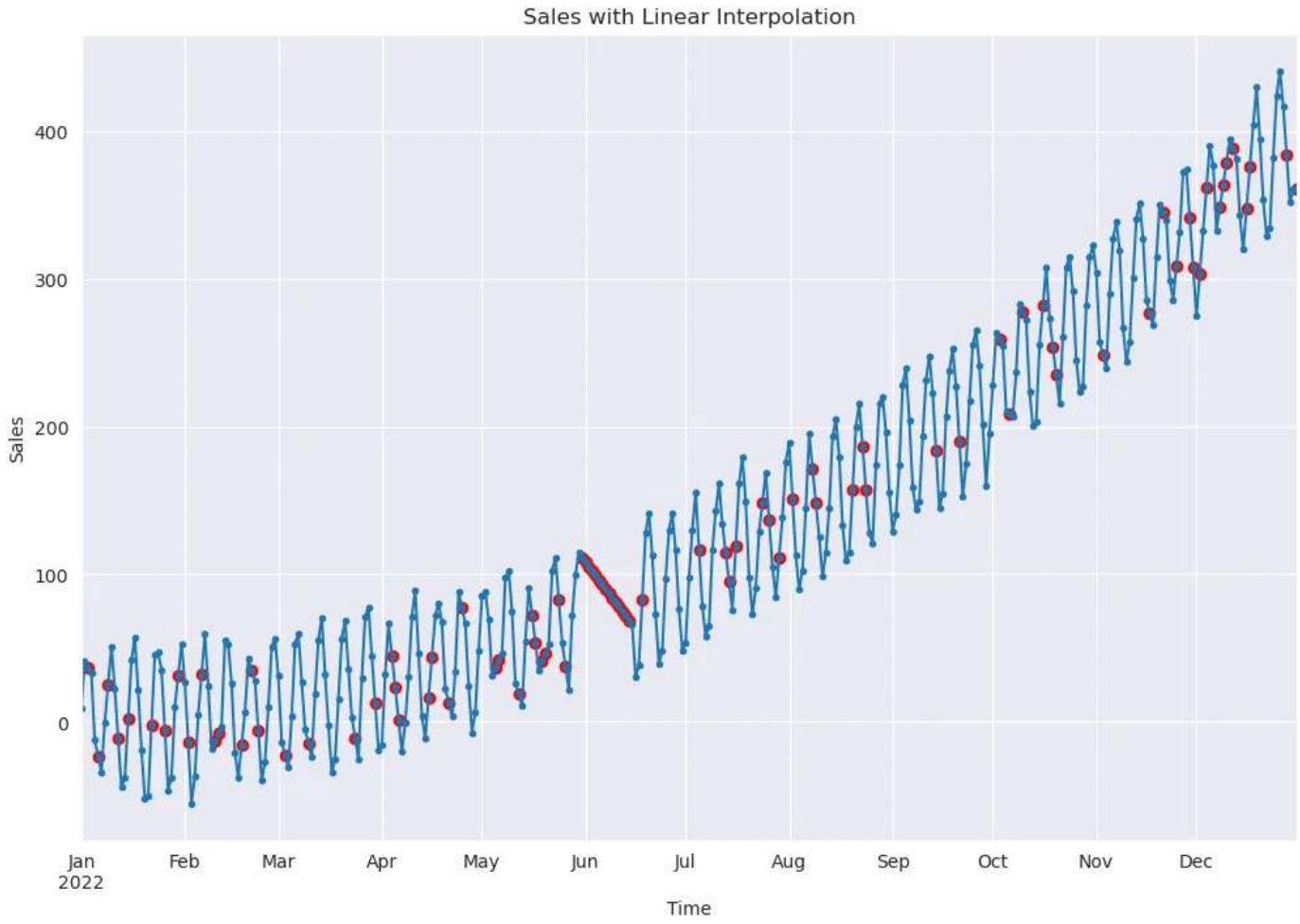
Impact on the Model-Building Process:

Using linear interpolation can provide a good estimate for the missing values when the data shows a linear trend, which can help in building a better model. However, if the data does not show a linear trend, this method can lead to poor model performance.

Let's see a code demo of linear interpolation.

```
# Apply the linear interpolation method
df_imputed = df.interpolate(method ='linear', limit_direction
='forward')

# Plot the main line with markers
df_imputed['sales'].plot(style='.-', figsize=(12,8), title='Sales with
Linear Interpolation')

# Add points where data was imputed with red color
```

```
imputed_indices = df[df['sales'].isnull()].index
plt.scatter(imputed_indices, df_imputed.loc[imputed_indices, 'sales'],
color='red')

# Set labels
plt.xlabel('Time')
plt.ylabel('Sales')

plt.show()
```



Spline Interpolation

Spline interpolation is a form of interpolation where the interpolant is a special type of piecewise polynomial called a spline. Spline interpolation is often preferred over polynomial interpolation because the interpolation error can be made small even when using low degree polynomials for the spline. Spline interpolation also does not suffer from the problem of Runge's phenomenon.

When should we use it?

Spline interpolation is used when the data is numeric and continuous, and you want to avoid the problem of overfitting that can occur with polynomial interpolation.

Pros

- Provides a smoother and more flexible fit than linear interpolation.
- Does not suffer from the problem of overfitting that can occur with polynomial interpolation.

Cons

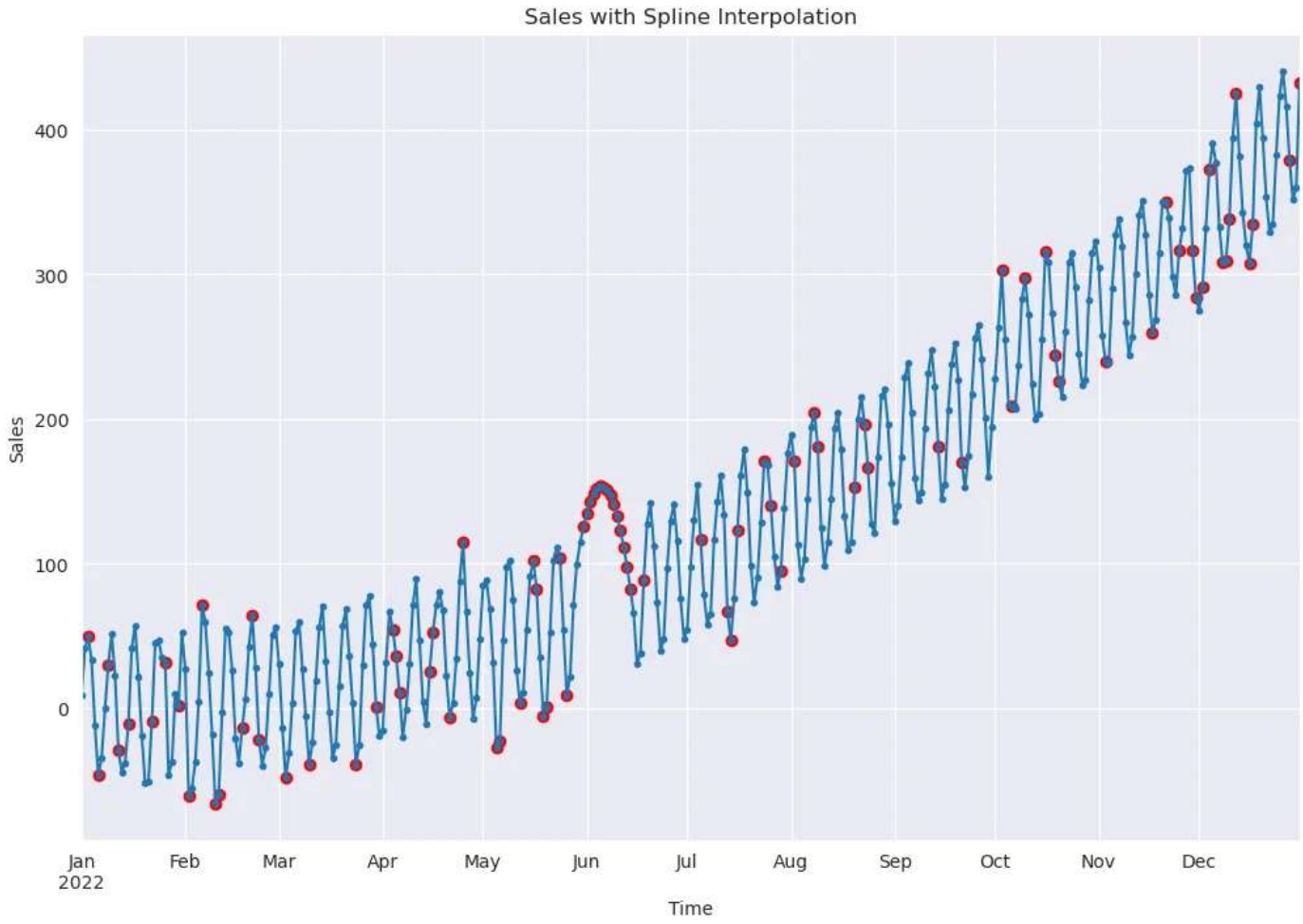
- More computationally intensive than linear interpolation.
- Can create unrealistic estimates if the data is not smooth.

Impact on the Model-Building Process:

Spline interpolation can provide a better fit to the data than linear interpolation, which can lead to more accurate models. However, if the data is not smooth, this method can lead to unrealistic estimates and poor model performance.

Let's see a code demo of spline interpolation.

```
# Apply the spline interpolation method
df_imputed = df.interpolate(method='spline',
limit_direction='forward', order=2)

# Plot the main line with markers
df_imputed['sales'].plot(style='.-', figsize=(12,8), title='Sales with
Spline Interpolation')

# Add points where data was imputed with red color
imputed_indices = df[df['sales'].isnull()].index
plt.scatter(imputed_indices, df_imputed.loc[imputed_indices, 'sales'],
color='red')

# Set labels
plt.xlabel('Time')
plt.ylabel('Sales')

plt.show()
```



As the graph clearly depicts, the imputation for the missing gap is totally unrealistic!

Regression Imputation

Regression imputation is a method of imputing missing values by using the relationships between the data. In this method, a regression model is used to predict the missing values based on other data.

When should we use it?

Regression imputation can be used when the data is numeric and there is a strong correlation between the variable with missing values and other variables.

Pros

- Makes use of the relationship between variables.
- Can be more accurate than mean, median or mode imputation.

Cons

- Can lead to an overestimate of the correlation between variables.

- The imputed values are estimates, not actual observations.

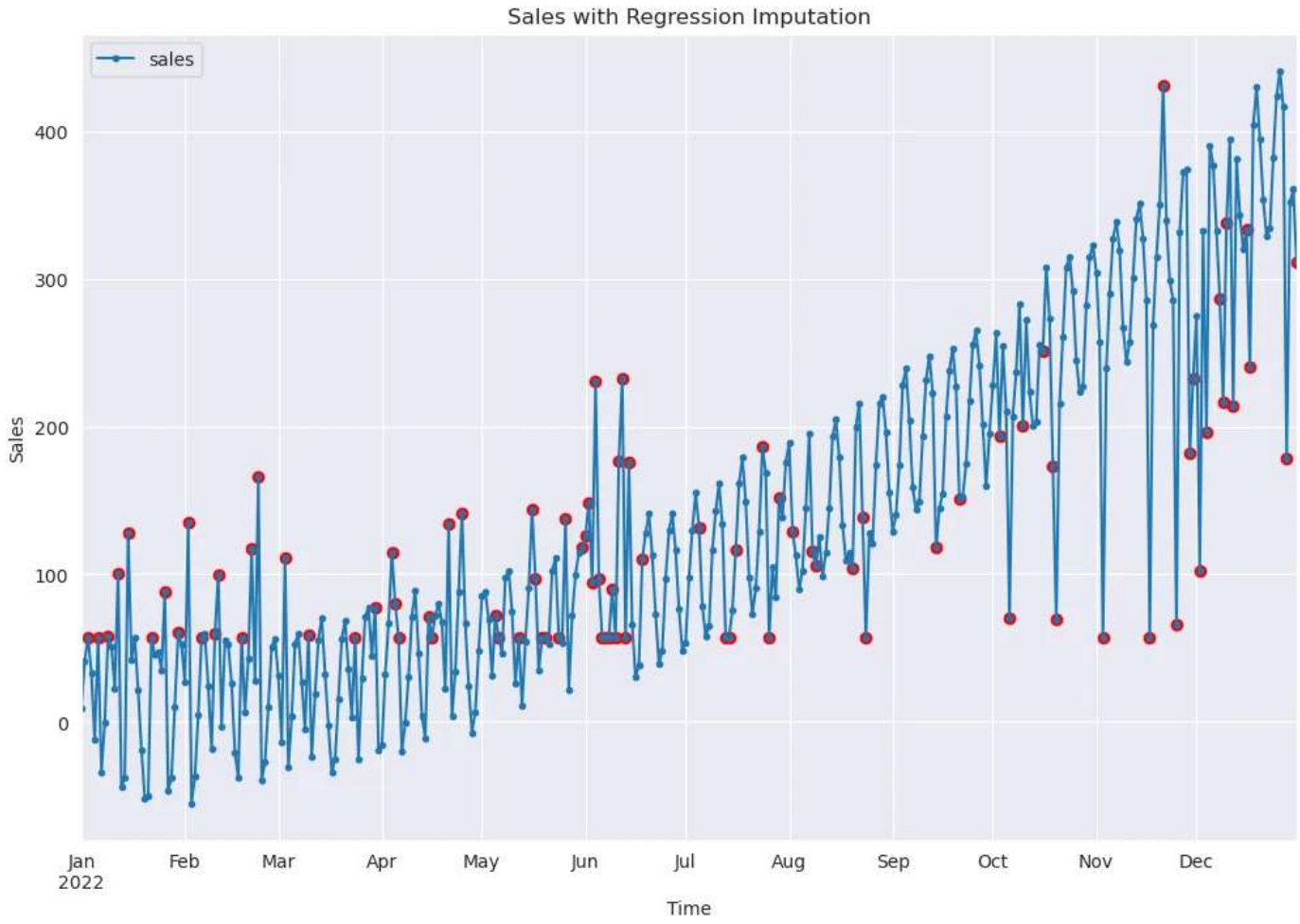
Impact on the Model-Building Process:

Regression imputation can lead to more accurate models by making use of the relationships between variables. However, it can also lead to an overestimate of the correlation between variables, which can affect the model's performance.

Let's see a code demo of regression imputation.

```
# Drop missing values to fit the regression model
df_imputed = df.copy()
df_non_missing = df.dropna()

# Instantiate the model
model = LinearRegression()

# Reshape data for model fitting (sklearn requires 2D array for
predictors)
X = df_non_missing['ad_spent'].values.reshape(-1, 1)
Y = df_non_missing['sales'].values

# Fit the model
model.fit(X, Y)

# Get indices of missing sales
missing_sales_indices = df_imputed[df_imputed['sales'].isnull()].index

# Predict missing sales values
predicted_sales = model.predict(df_imputed.loc[missing_sales_indices,
'ad_spent'].values.reshape(-1, 1))

# Fill missing sales with predicted values
df_imputed.loc[missing_sales_indices, 'sales'] = predicted_sales

# Plot the main line with markers
df_imputed[['sales']].plot(style='.-', figsize=(12,8), title='Sales
with Regression Imputation')

# Add points where data was imputed with red color
plt.scatter(missing_sales_indices, predicted_sales, color='red',
label='Regression Imputation')

# Set labels
plt.xlabel('Time')
plt.ylabel('Sales')
```

```
plt.show()
```



K-Nearest Neighbors (KNN) Imputation

K-Nearest Neighbors (KNN) is a machine learning algorithm that is used for classification and regression problems. However, it can also be used for imputing missing data. The KNN imputation method works by finding the K-nearest neighbors to the observation with missing data and then imputing them based on the non-missing values in the neighbors.

When should we use it?

KNN can be used when the data are missing at random, and the variables are almost equally important. KNN is better when the data are uniformly distributed, and it's not suitable for high dimensional data.

Pros

- KNN is simple and easy to implement.

- It doesn't make any assumptions about the data.
- It can be much more accurate than other methods, especially if the data are uniformly distributed.

Cons

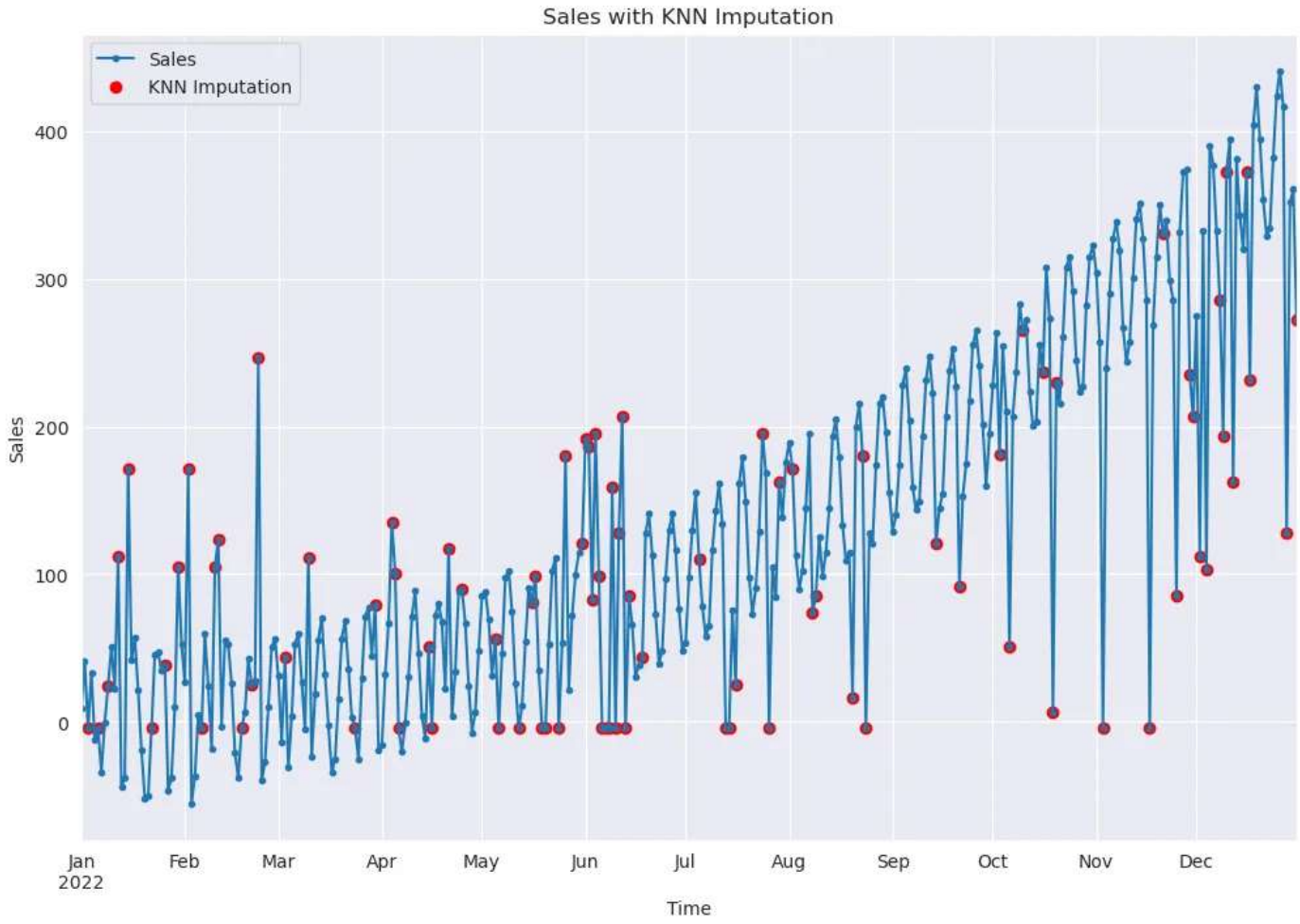
- KNN can be computationally expensive, especially if the dataset is large with many features.
- It's sensitive to the scale of the data and irrelevant features.
- It doesn't handle categorical data well.

Impact on the Model-Building Process:

KNN imputation can introduce bias into the data if the missing data mechanism isn't random. This is because the KNN algorithm relies on the assumption that similar observations exist in the dataset. If this assumption is violated, the imputed values may be far from the true values, leading to biased estimates in the model-building process.

```
# Initialize the KNN imputer with k=5
imputer = KNNImputer(n_neighbors=3)

# Apply the KNN imputer
# Note: the KNNImputer requires 2D array-like input, hence the double
brackets.
df_imputed = df.copy()
df_imputed[['sales', 'ad_spent']] =
imputer.fit_transform(df_imputed[['sales', 'ad_spent']])

# Create a matplotlib plot
plt.figure(figsize=(12,8))
df_imputed['sales'].plot(style='.-', label='Sales')

# Add points where data was imputed
imputed_indices = df[df['sales'].isnull()].index
plt.scatter(imputed_indices, df_imputed.loc[imputed_indices, 'sales'],
color='red', label='KNN Imputation')

# Set title and labels
plt.title('Sales with KNN Imputation')
plt.xlabel('Time')
plt.ylabel('Sales')
```

```
plt.legend()
plt.show()
```



Seasonal Trend Decomposition using Loess (STL) Imputation

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 for imputing missing data in a time series. In the STL imputation method, the missing values are initially estimated via interpolation to allow for STL decomposition. Afterward, 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.

When should we use it?

STL imputation can be used when dealing with time series data that exhibits a seasonal pattern. It is particularly useful when the data are missing at random and the missingness is not related to the trend or seasonality of the time series.

Pros

- STL imputation can capture complex seasonal patterns that other imputation methods might miss.
- It can handle any type of seasonality, not just monthly or quarterly.
- It is flexible, as the user can specify the amount of smoothing.

Cons

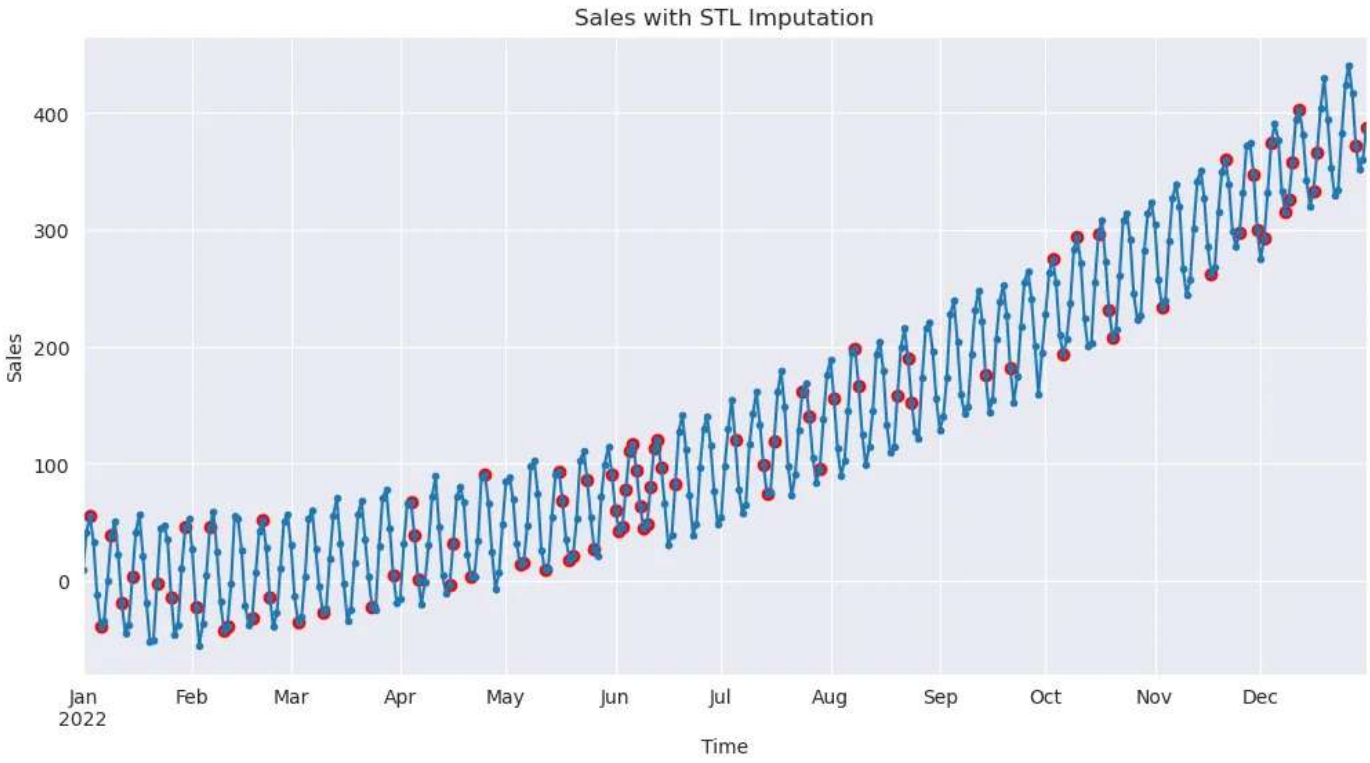
- STL imputation might not be appropriate if there is a high level of noise in the data.
- It requires a sufficient number of complete cycles in the time series to accurately estimate the seasonal component.
- The method might introduce some bias in the presence of trend-cycle interactions.

Impact on the Model-Building Process:

STL imputation aims to preserve the overall structure of the time series, particularly its trend and seasonal components, when filling in missing values. The imputed values are therefore more likely to reflect the actual dynamics of the time series, leading to more accurate models. However, as with any imputation method, caution should be exercised when the amount of missing data is large, as this could potentially lead to overfitting or biased results in subsequent analyses.

```
# Make a copy of the original dataframe
df_copy = df.copy()

# Fill missing values in the time series
imputed_indices = df[df['sales'].isnull()].index
# Apply STL decompostion
stl = STL(df_copy['sales'].interpolate(), seasonal=31)
res = stl.fit()

# Extract the seasonal and trend components
seasonal_component = res.seasonal

# Create the deseasonalised series
df_deseasonalised = df_copy['sales'] - seasonal_component

# Interpolate missing values in the deseasonalised series
df_deseasonalised_imputed =
df_deseasonalised.interpolate(method="linear")
```

```
# Add the seasonal component back to create the final imputed series
df_imputed = df_deseasonalised_imputed + seasonal_component

# Update the original dataframe with the imputed values
df_copy.loc[imputed_indices, 'sales'] = df_imputed[imputed_indices]

# Plot the series using pandas
plt.figure(figsize=[12, 6])
df_copy['sales'].plot(style='.-',  label='Sales')
plt.scatter(imputed_indices, df_copy.loc[imputed_indices, 'sales'],
color='red')

plt.title("Sales with STL Imputation")
plt.ylabel("Sales")
plt.xlabel("Time")
plt.show()
```



About Me

Hello, I'm **Ahmed Abulkhair**, a Senior Data Scientist at Seamlabs. I love sharing my knowledge and experiences. Through my work, I aim to simplify complex machine learning concepts and make them accessible for everyone.

Here's where you can find and support my work:

- **Medium:** I publish my articles on Medium. If you find them useful, don't hesitate to give them a clap!
- **LinkedIn:** Connect with me on LinkedIn to stay updated with my professional activities and insights.
- **Kaggle:** I'm an Expert on Kaggle. Follow me there to see my data science projects and solutions.

Your support through follows, claps on Medium, and upvotes on Kaggle is greatly appreciated and encourages me to continue creating and sharing valuable content. Let's learn and grow together!

Time Series Forecasting | Time Series Analysis | Machine Learning | Missing Data | Trending

Follow

Written by Ahmed Abulkhair

239 Followers

Lead Data Scientist @ ShopBrain | Love to share! | LinkedIn: https://www.linkedin.com/in/aaabulkhair/ | Topmate: https://topmate.io/abulkhair

More from Ahmed Abulkhair