Highly Integrated System Project

Assignment 2

Aaliya Javed
Ali Mohammad Nekkoh
Abu Saleh Md Nayem
Mohsina Binte Asad
Zain Hanif

November 4, 2024

Introduction to Practical Time Series Analysis

In the previous chapter, we introduced the concept of a time series and established some standard notations and terminologies. In this chapter, we will shift our focus from theoretical concepts to practical applications. We will begin working with actual time series data, exploring various datasets, and learning techniques essential for data processing and analysis. Although time series data is ubiquitous, we are about to explore and practice with some sample datasets. Our goal is to preprocess this data correctly and learn how to address missing values effectively. This chapter will cover the following key topics:

- Understanding the time series dataset
- Refresher on `pandas` datetime operations, indexing, and slicing
- Techniques for handling missing data
- Mapping additional information to time series data
- Saving and loading files to and from disk
- Strategies for handling longer periods of missing data

Technical Requirements

To execute the code in this chapter, you will need to set up the Anaconda environment as outlined in the Preface. This setup will ensure you have all the necessary packages and datasets.

Handling Time Series Data

Working with time series data is similar to handling other tabular datasets, but it places a greater emphasis on the temporal aspect. Luckily, `pandas` provides excellent support for time series operations. Let's get hands-on and explore a dataset from scratch. Throughout this book, we'll use the London Smart Meters dataset. If you haven't downloaded the data yet, please refer to the instructions in the Preface.

Understanding the Time Series Dataset

Before diving into Exploratory Data Analysis (EDA) — which we will cover in Chapter 3, *Analyzing and Visualizing Time Series Data* — it is essential to understand the dataset. This means knowing where the data comes from, how it was generated, and the domain context.

The London Smart Meters dataset comes from the *London Data Store*, an open data platform. It was collected and enriched by Jean-Michel D and is available on Kaggle.

Dataset Overview

The dataset contains energy consumption readings from 5,567 London households involved in the Low Carbon London project, managed by UK Power Networks. The data covers November 2011 to February

2014, with readings taken at half-hourly intervals. It also includes valuable metadata about the households.

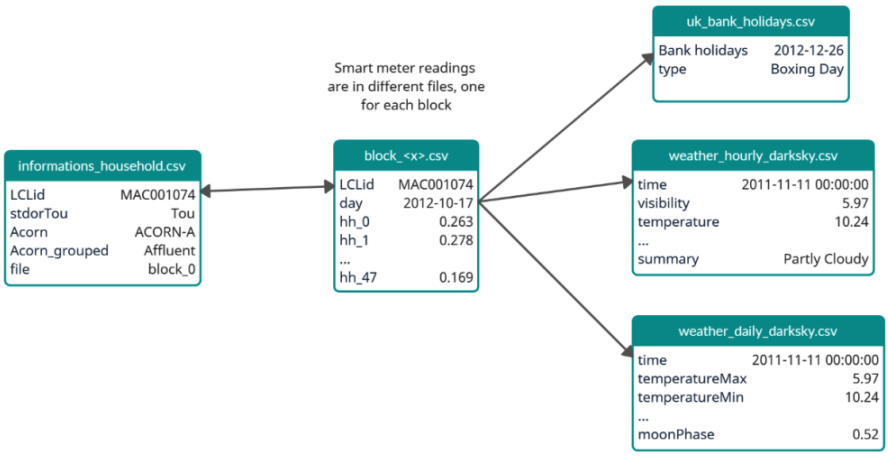

Figure 1: Data Model

Key Metadata Details

- **Acorn Classification**: CACI UK has categorized the UK population into demographic types called *Acorn*. Each household in the dataset has an Acorn classification. Examples include *Lavish Lifestyles, City Sophisticates, Student Life*, and others. These types are grouped into broader categories like *Affluent Achievers, Rising Prosperity, Financially Stretched*. You can find a full list of Acorn classes in Figure 2. More details are available in the Acorn user guide: https://acorn.caci.co.uk/downloads/Acorn-User-guide.pdf.
- **Customer Groups**: The dataset includes two main groups:
  - *Dynamic Time-of-Use (dToU)*: Customers subjected to dynamic pricing in 2013, with prices communicated a day in advance via Smart Meter IHD or text message.
  - *Flat-Rate Tariffs*: Customers who were on standard, flat-rate pricing.
- **Enriched Data**: Additional information such as weather data and UK bank holidays has been added to the dataset by Jean-Michel D.

ACORN Group	ACORN Class
Affluent Achievers	A–Lavish Lifestyles
	B–Executive Wealth
	C–Mature Money
Rising Prosperity	D–City Sophisticates
	E–Career Climbers
Comfortable Communities	F–Countryside Communities
	G–Successful Suburbs
	H–Steady Neighborhoods
	I–Comfortable Seniors
	J–Starting Out
Financially Stretched	K–Student Life
	L–Modest Means
	M–Striving Families
	N–Poorer Pensioners
Urban Adversity	O–Young Hardship
	P–Struggling Estates
	Q–Difficult Circumstances

Figure 2: ACORN Classification

Pandas DateTime Operations, Indexing, and Slicing - A Refresher

Instead of using a complex dataset, we will use a simple and well-organized stock exchange price dataset from the UCI Machine Learning Repository to explore the features of pandas.

We can load the dataset with the following code:

```
df = pd.read_excel("https://archive.ics.uci.edu/ml/machine-learning-databases/00247/data_akbilgic.xlsx", skip
```

The DataFrame we read looks like this:

	date	ISE	ISE.1	SP	DAX	FTSE	NIKKEI	BOVESPA	EU	EM
0	2009-01-05	0.035754	0.038376	-0.004679	0.002193	0.003894	0.000000	0.031190	0.012698	0.028524
1	2009-01-06	0.025426	0.031813	0.007787	0.008455	0.012866	0.004162	0.018920	0.011341	0.008773
2	2009-01-07	-0.028862	-0.026353	-0.030469	-0.017833	-0.028735	0.017293	-0.035899	-0.017073	-0.020015
3	2009-01-08	-0.062208	-0.084716	0.003391	-0.011726	-0.000466	-0.040061	0.028283	-0.005561	-0.019424
4	2009-01-09	0.009860	0.009658	-0.021533	-0.019873	-0.012710	-0.004474	-0.009764	-0.010989	-0.007802

Figure 3: The Dataframe with stock exchange prices

Now that we have loaded the DataFrame, we can start manipulating it.

Converting Date Columns into `pd.Timestamp`/`DatetimeIndex`

To convert a date column into pandas datetime format, we use the function `pd.to_datetime`. This function automatically infers the datetime format and converts the input into a `pd.Timestamp` for a single date string or a `DatetimeIndex` for a list of date strings.

For example, when we pass a single date string:

```
>>> pd.to_datetime("13-4-1987").strftime("%d, %B %Y")
'13, April 1987'
```

However, automatic parsing can fail in some cases. For instance, if we try to convert the date January 4, 1987, like this:

```
>>> pd.to_datetime("4-1-1987").strftime("%d, %B %Y")
'01, April 1987'
```

This output is unexpected because pandas assumes the month comes first. To specify that the day comes first, we can use the `dayfirst=True` argument:

```
>>> pd.to_datetime("4-1-1987", dayfirst=True).strftime("%d, %B %Y")
'04, January 1987'
```

Another issue arises with non-standard date formats. In such cases, we can provide a formatted string to guide pandas in parsing the date correctly:

```
>>> pd.to_datetime("4 | 1 | 1987", format="%d | %m | %Y").strftime("%d, %B %Y")
'04, January 1987'
```

For a full list of `strftime` conventions, visit `https://strftime.org/`.

Practitioner's Tip on Date Handling in Pandas

Due to the variety of data formats, pandas may incorrectly infer dates when reading a file, leading to potential errors. To manage this behavior, we can use several options:

- Use the `parse_dates` flag to disable automatic date parsing.
- Use the `date_parser` argument to pass a custom date parser.
- Utilize the `year_first` and `dayfirst` arguments to clarify date formats.

I recommend setting `parse_dates=False` when using `pd.read_csv` or `pd.read_excel` to prevent automatic parsing. After that, you can convert the date using the `format` parameter, which allows you to specify the date format explicitly using `strftime` conventions.

For example, to convert the date column in our stock prices dataset, you can use:

```
df['date'] = pd.to_datetime(df['date'], yearfirst=True)
```

After conversion, the 'date' column's data type should be either `datetime64[ns]` or `<M8[ns]`, which are native datetime formats in pandas/NumPy. This conversion unlocks a range of functionalities. For instance, you can now use the `min()` and `max()` functions:

```
>>> df.date.min(), df.date.max()
(Timestamp('2009-01-05 00:00:00'), Timestamp('2011-02-22 00:00:00'))
```

Using the `.dt` Accessor and Datetime Properties

With the date column in datetime format, you can access various properties using the `.dt` accessor. Here's an example of retrieving some properties:

```
print(f"""
Date: {df.date.iloc[0]}
Day of year: {df.date.dt.day_of_year.iloc[0]}
Day of week: {df.date.dt.dayofweek.iloc[0]}
Week of Year: {df.date.dt.isocalendar().week.iloc[0]}
Month: {df.date.dt.month.iloc[0]}
Month Name: {df.date.dt.month_name().iloc[0]}
Quarter: {df.date.dt.quarter.iloc[0]}
Year: {df.date.dt.year.iloc[0]}
ISO Week: {df.date.dt.isocalendar().week.iloc[0]}
""")
```

As of pandas 1.1.0, `week_of_year` has been deprecated due to inconsistencies at year-end, replaced by the ISO calendar standards.

Slicing and Indexing

The functionality improves when you set the date column as the DataFrame's index. This allows for advanced slicing operations along the datetime axis. For instance:

```
# Setting the index as the datetime column
df.set_index("date", inplace=True)

# Select all data after 2010-01-04 (inclusive)
df["2010-01-04":]

# Select data between 2010-01-04 and 2010-02-06 (exclusive)
df["2010-01-04":"2010-02-06"]

# Select data for the year 2010 and before
df[:"2010"]

# Select data between January and June 2010 (inclusive)
df["2010-01":"2010-06"]
```

Creating Date Sequences and Managing Date Offsets

If you're familiar with the `range` function in Python or `np.arange` in NumPy, you know they help create integer or float sequences by defining a start and an end point. Pandas offers a similar functionality for datetime with the `pd.date_range` function. This function accepts start and end dates, as well as a frequency (daily, monthly, etc.), to create a sequence of dates.

Here are a few examples of how to create sequences of dates using `pd.date_range`:

1. Specifying Start and End Dates with Frequency

To create a date range from a start to an end date with a specified frequency, you can use:

```
pd.date_range(start="2018-01-20", end="2018-01-23", freq="D").astype(str).tolist()
```

Output: `['2018-01-20', '2018-01-21', '2018-01-22', '2018-01-23']`

2. Specifying Start and Number of Periods

You can also specify a start date and the number of periods to generate:

```
pd.date_range(start="2018-01-20", periods=4, freq="D").astype(str).tolist()
```

Output: `['2018-01-20', '2018-01-21', '2018-01-22', '2018-01-23']`

3. Generating a Date Sequence Every 2 Days

To generate a sequence of dates at a specified interval (e.g., every 2 days):

```
pd.date_range(start="2018-01-20", periods=4, freq="2D").astype(str).tolist()
```

Output: ['2018-01-20', '2018-01-22', '2018-01-24', '2018-01-26']

4. Generating a Date Sequence Every Month (Month End)

By default, `pd.date_range` generates dates at the end of the month:

```
pd.date_range(start="2018-01-20", periods=4, freq="M").astype(str).tolist()
```

Output: ['2018-01-31', '2018-02-28', '2018-03-31', '2018-04-30']

5. Generating a Date Sequence Every Month (Month Start)

To generate a sequence of dates at the beginning of each month, use the `MS` (Month Start) frequency:

```
pd.date_range(start="2018-01-20", periods=4, freq="MS").astype(str).tolist()
```

Output: ['2018-02-01', '2018-03-01', '2018-04-01', '2018-05-01']

6. Adding Days to a Date Range

To add a specific number of days to a date range:

```
(pd.date_range(start="2018-01-20", end="2018-01-23", freq="D") + pd.Timedelta(4, unit="D")).astype(str).tolis
```

Output: ['2018-01-24', '2018-01-25', '2018-01-26', '2018-01-27']

7. Adding Weeks to a Date Range

To add weeks to a date range, you can similarly use:

```
(pd.date_range(start="2018-01-20", end="2018-01-23", freq="D") + pd.Timedelta(4, unit="W")).astype(str).tolis
```

Output: ['2018-02-17', '2018-02-18', '2018-02-19', '2018-02-20']

Handling Missing Data

When working with large datasets, encountering missing data is common. Before filling in missing values with means or dropping rows, it's important to consider a few aspects:

- **Understanding the Missing Data:** Consider whether the missing data is truly missing or if it provides valuable information. This involves understanding the Data Generating Process (DGP) behind your data.
- **Example:** If you are analyzing sales data from a local supermarket and find that certain products have no transactions for some days, you need to decide if the missing data indicates zero sales for those days rather than an error. In this case, it may be appropriate to fill the missing entries with zeros instead of dropping them or imputing them with a mean value.

Handling Patterns of Missing Data

When analyzing missing data in your dataset, it's crucial to identify patterns, as this affects how you manage the gaps.

1. Patterns in Missing Data

- If you find that data is missing every Sunday, it indicates a specific pattern. This situation can complicate how you handle those missing values:

- Filling these gaps with zeros can mislead models that predict future values based on recent data, especially on Mondays.
- Instead, if you indicate to the model that the previous day was Sunday, it can learn from this context, potentially improving accuracy.

2. Anomalous Zero Sales

- Zero sales for a normally high-selling product may be due to various reasons, such as:

- Malfunctions in the POS machine
- Data entry errors
- Out-of-stock situations

- These missing values can often be addressed using various imputation techniques, which will be discussed in later sections.

Practitioner's Tip

When using methods like `read_csv`, pandas offers useful options to manage missing values:

- Pandas automatically considers many values (like #N/A and null) as NaN.
- You can customize which values are treated as NaN by using the `na_values` and `keep_default_na` parameters.

Example Dataset

For illustration, we can analyze PM2.5 readings from the Monash region, which includes some missing values. This dataset is published by the ACT Government of Canberra, Australia, under the CC BY Attribution 4.0 International License:

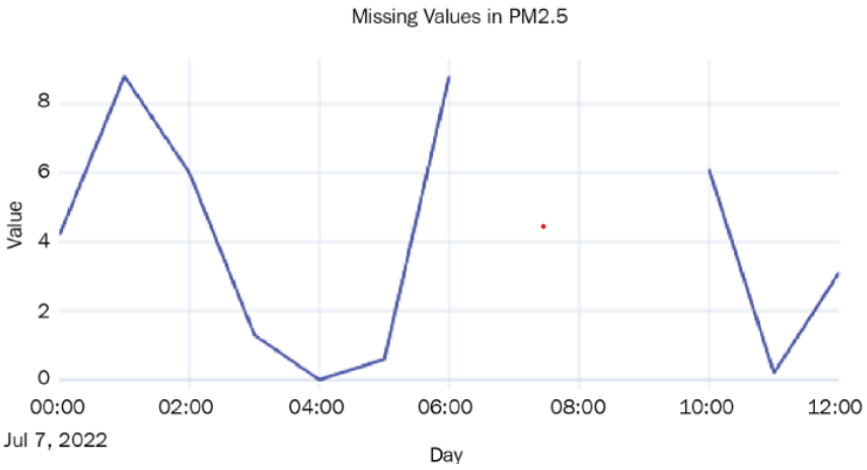

Figure 4: Missing values in the Air Quality dataset

Techniques for Filling Missing Values

In data analysis, it is common to encounter missing values in datasets. Here are three simple techniques for handling these missing values in pandas:

1. Last Observation Carried Forward (Forward Fill)

This technique fills missing values by carrying forward the last observed value. It continues to use this value until a new observation is encountered. In pandas, this can be done using:

```
df['pm2_5_1_hr'].ffill()
```

2. Next Observation Carried Backward (Backward Fill)

This technique fills missing values by taking the next available observation and using it to fill the preceding missing values. This is done with:

```
df['pm2_5_1_hr'].bfill()
```

3. Mean Value Fill

In this technique, we calculate the mean of the entire series and use this value to fill in any missing entries. The implementation in pandas looks like this:

```
df['pm2_5_1_hr'].fillna(df['pm2_5_1_hr'].mean())
```

Visualizing Imputed Values

To better understand the effects of these imputation techniques, we can plot the imputed lines generated by each method. Below is a placeholder for including a plot of the results:

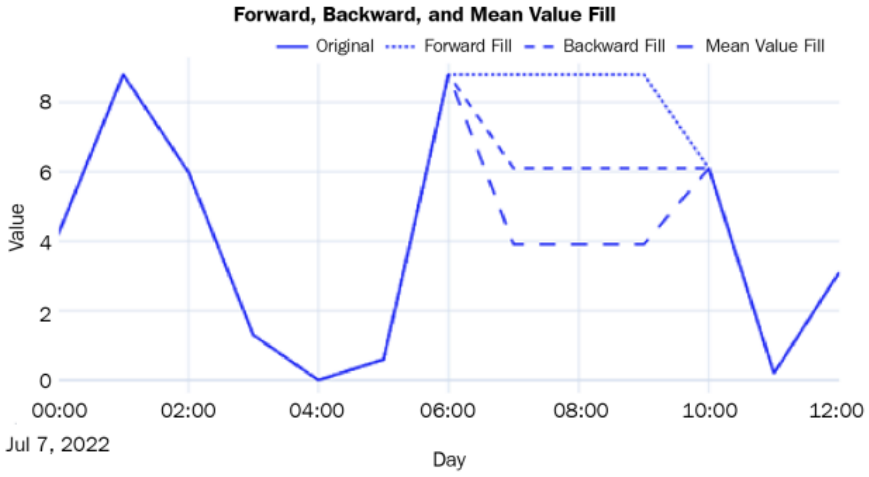

Figure 5: Imputed lines using Forward Fill, Backward Fill, and Mean Value Fill

Interpolation Techniques for Imputing Missing Values

Interpolation is another powerful family of techniques for filling in missing values. Here are two common methods:

1. Linear Interpolation

Linear interpolation estimates missing values by drawing a straight line between two observed points and filling the gaps with values that lie on this line. In pandas, you can perform linear interpolation as follows:

```
df['pm2_5_1_hr'].interpolate(method="linear")
```

2. Nearest Interpolation

Nearest interpolation fills missing values by finding the closest observed value (either preceding or following) and using that value to fill in the gaps. This method can be seen as a combination of forward and backward fill. In pandas, it is implemented like this:

```
df['pm2_5_1_hr'].interpolate(method="nearest")
```

Visualizing Interpolated Values

To illustrate the effects of these interpolation methods, we can plot the results of both linear and nearest interpolation. Below is a placeholder for including the plot:

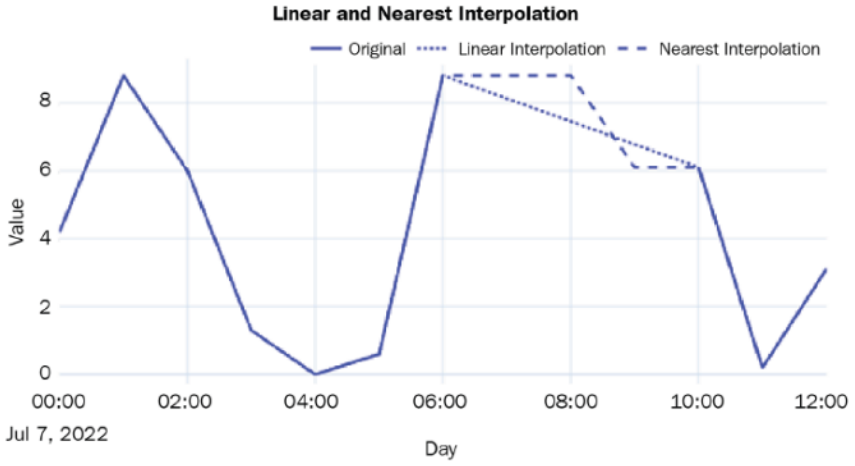

Figure 6: Interpolated lines using Linear Interpolation and Nearest Interpolation

Non-Linear Interpolation Techniques

In addition to linear interpolation, there are several non-linear interpolation techniques available in pandas. These methods use SciPy routines in the backend to achieve more flexible fits to the data. Two common non-linear interpolation methods are Spline and Polynomial interpolations.

Spline Interpolation

Spline interpolation fits a spline function to the observed data points, allowing for a smoother curve that can capture the underlying trends more effectively than linear interpolation. You can perform spline interpolation in pandas as follows:

```
df['pm2_5_1_hr'].interpolate(method="spline", order=2)
```

In this example, the `order` parameter specifies the order of the spline. An order of 2 corresponds to a quadratic spline.

Polynomial Interpolation

Polynomial interpolation works similarly to spline interpolation but fits a polynomial function of a specified order to the data. This method can also capture non-linear trends in the data. Here's how to perform polynomial interpolation in pandas:

```
df['pm2_5_1_hr'].interpolate(method="polynomial", order=5)
```

In this case, the `order` parameter specifies the degree of the polynomial used to fit the data. A higher order provides greater flexibility, but it may also lead to overfitting. When using non-linear interpolation methods, it's important to choose the order carefully. Higher-order functions can fit the data closely but may lead to oscillations that do not reflect the underlying trend, especially with sparse data.

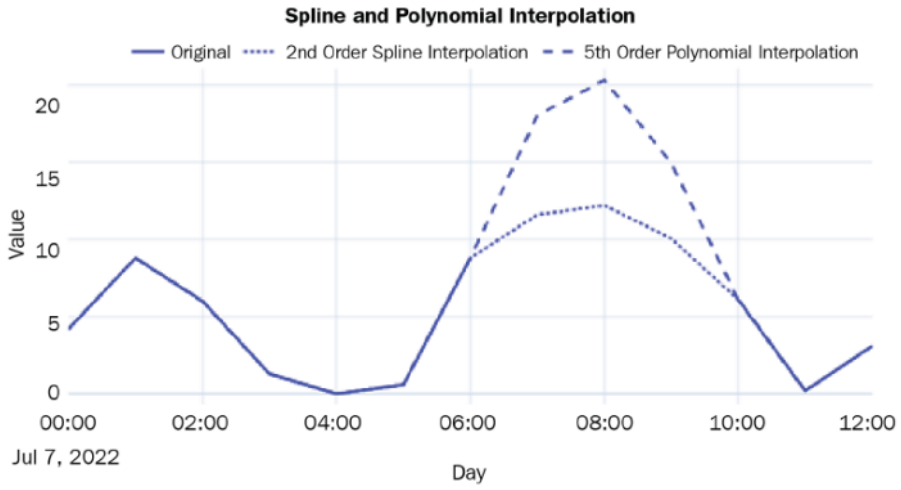

Figure 7: Interpolated missing values using Spline and Polynomial

Before processing half-hourly block-level data into time series data, it's important to understand the key types of information in a time series dataset:

- **Time Series Identifiers:** Unique identifiers for each time series, like SKU names, retail sales IDs, or consumer IDs in the energy dataset.
- **Metadata or Static Features:** Information that remains constant over time, such as household classifications (e.g., ACORN classification).
- **Time-Varying Features:** Information that changes over time, like weather data, which varies at each time point.

Next, we consider the formatting of time series datasets, which can be arranged in several ways:

- **Compact Form:** Each time series is represented by a single row in a DataFrame, where time is managed as an array within that row. The identifiers and metadata are stored in columns, with time-varying features also in column form as arrays. The DataFrame includes `start_datetime` and `frequency` columns to manage time. This approach is memory-efficient and speeds up processing for regularly sampled time series.

LCLid	start_timestamp	frequency	energy_consumption	series_length
MAC000002	2012-10-13	30min	[0.263, 0.2689999999999999, 0.275, 0.256, 0.21...	24144
MAC000246	2011-12-04	30min	[0.175, 0.098, 0.144, 0.065, 0.071, 0.037, 0.0...	39216
MAC000450	2012-03-23	30min	[0.337, 1.426, 0.996, 0.971, 0.994, 0.952, 0.8...	33936
MAC001074	2012-05-09	30min	[0.18, 0.086, 0.106, 0.173, 0.146, 0.223, 0.21...	31680
MAC003223	2012-09-18	30min	[0.076, 0.079, 0.123, 0.109, 0.051, 0.069, 0.0...	25344

Figure 8: Compact Form Data

The **expanded form** of time series data is structured so that each time series is represented by multiple rows in a DataFrame. If there are n time steps in the series, it will take up n rows, with the time series

identifiers and metadata repeated for each row. Time-varying features are also expanded along the rows, and instead of having start date and frequency, the DataFrame includes a `datetime` column.

For example, an expanded dataset might look like this:

In this case, if the compact form used a time series identifier as a key, the combination of the time series identifier and the `datetime` column would form the new key.

LCLid	energy_consumption	series_length	timestamp	frequency
MAC000002	0.263	24144	2012-10-13 00:00:00	30min
MAC000002	0.269	24144	2012-10-13 00:30:00	30min
MAC000002	0.275	24144	2012-10-13 01:00:00	30min
MAC000002	0.256	24144	2012-10-13 01:30:00	30min
MAC000002	0.211	24144	2012-10-13 02:00:00	30min

Figure 9: Expended Form Data

Wide-format data is a more traditional approach where the date is an index or column and different time series are represented in separate columns. As more time series are added, the dataset becomes wider. This format is limited because it doesn't allow for including additional metadata about the time series, such as pricing details. Additionally, wide formats are not well-suited for relational databases since new columns must be added for each new time series. This book will not use the wide format.

When working with time series data, it's crucial to ensure that the data is collected at regular time intervals. Even if the data is intended to be regularly sampled, there can be missing samples due to errors in data collection or other issues. Therefore, we need to enforce regular intervals in the time series.

Best Practice: When dealing with datasets containing multiple time series, check that all time series have the same end date. If they don't, align them to the latest date found across the dataset.

In our smart meters dataset, some entries in the LCLid column may end earlier than others. This could happen if a household opted out of the program or moved out. We must handle these variations while ensuring regular intervals.

Converting the London Smart Meters dataset into a time series format

Next, we will learn how to convert the dataset into a proper time series format. The code for this process can be found in the notebook titled `"02 - Preprocessing London Smart Meter Dataset.ipynb"`.

To convert the London Smart Meters dataset into a time series format, the steps will vary based on the original data structure. Here, we will focus on transforming the dataset and apply these lessons to other datasets.

Before processing the data into either a compact or expanded form, we need to complete two steps:

1. **Find the Global End Date:** Determine the maximum date across all block files to identify the global end date for the time series.

2. **Basic Preprocessing:** The `hhblock_dataset` has a structure where each row contains a date, and the columns represent half-hourly blocks. We need to reshape this into a long format, where each row has a date and a single half-hourly block, making it easier to work with.

Next, we will define separate functions for converting the data into compact and expanded forms. These functions will be applied to each `LCLid` column individually, as each one may have a different start date.

For the **expanded form**, the function will:

1. Find the start date for the specific `LCLid`.
2. Create a standard DataFrame using the start date and the global end date.
3. Left merge the `LCLid` DataFrame with the standard DataFrame, filling in any missing data with `np.nan`.
4. Return the merged DataFrame.

After processing each `LCLid` DataFrame into the expanded form, we need to take a few more steps:

1. **Concatenate** all the DataFrames into a single DataFrame.
2. **Create an `offset` column** that represents the half-hour blocks numerically (e.g., `hh_3` becomes `3`).
3. **Create a `timestamp`** by adding a 30-minute offset to the date and then drop any unnecessary columns.

For one block of data, this expanded representation uses about 47 MB of memory.

For the **compact form**, the conversion function does the following:

1. Finds the **start date** and the **time series identifiers**.
2. Creates a **standard DataFrame** using the start date and the global end date.
3. **Left merges** the `LCLid` DataFrame with the standard DataFrame, filling missing data with `np.nan`.
4. **Sorts the values** by date.
5. Returns the **time series array**, along with the time series identifier, start date, and the length of the time series.

Mapping Additional Information

The dataset comprises three essential files for enriching the time series data:

- **Household Information** (`informations_households.csv`): Contains metadata about each household, including static features. This file is merged based on the time series identifier (`LCLid`).
- **Bank Holidays** (`uk_bank_holidays.csv`): Lists holiday dates and types affecting energy consumption patterns. It requires alignment with the time series by resampling and forward-filling for 30-minute intervals.
- **Weather Data** (`weather_hourly_darksky.csv`): Contains daily weather data, which must be downsampled to match the time series' 30-minute interval.

Best Practice for Merging

- When merging dataframes using pandas, you might find that the number of rows changes unexpectedly. This can happen if there are duplicates in the keys used for merging.
- To avoid this, use the `validate` parameter in the `merge` function. This helps ensure that the merge behaves as expected and alerts you if there are issues.

Data Processing Steps

1. **Bank Holidays**:
   - Convert dates to `datetime`, set as index.
   - Resample to a 30-minute frequency, forward-fill, and replace remaining NaN values with `NO_HOLIDAY`.
2. **Weather Data**:
   - Downsample to a 30-minute frequency and forward-fill the weather features to fill the missing values that were created while resampling.

Saving and Loading Large Datasets

The compact DataFrame is approximately 10 MB. However, saving it in formats like CSV or Pickle may expand its size. Use `parquet` format and chunk the data for efficient compression, retaining data types, and facilitating quick iterations.

Handling Longer Periods of Missing Data

When dealing with missing data, simple methods like **forward filling**, **backward filling**, and **interpolation** work well for small gaps. However, these techniques struggle when there are **large sections** of data missing.

To start, we load specific data blocks from a parquet file:

```
block_df = pd.read_parquet("data/london_smart_meters/preprocessed/london_smart_meters_merged_block_0-7.parque
```

This data is in **compact form**, which we need to convert into **expanded form** for easier analysis. For demonstration purposes, we will extract one block.

Converting to Expanded Form

We can use a function called `compact_to_expanded` to convert the data:

```
exp_block_df = compact_to_expanded(
    block_df[block_df.file == "block_7"],
    timeseries_col='energy consumption',
```

```
    static_cols=["frequency", "series_length", "stdorTou", "Acorn", "Acorn_grouped", "file"],
    time_varying_cols=[
        'holidays', 'visibility', 'windBearing', 'temperature',
        'dewPoint', 'pressure', 'apparentTemperature', 'windSpeed',
        'precipType', 'icon', 'humidity', 'summary'
    ],
    ts_identifier="LCLid"
)
```

Visualizing Missing Data

A useful tool for visualizing missing data in related time series is the **missingno** library. Here's how to set it up:

1. **Pivot the Data**: Arrange the data with the timestamp as the index and each time series as columns.

```
plot_df = pd.pivot_table(exp_block_df, index="timestamp", columns="LCLid", values="energy_consumptio
```

2. **Generate the Plot**: Use the `msno.matrix` function to create a visualization, specifying the frequency for the X-axis.

```
msno.matrix(plot_df, freq="M")
```

The preceding code produces the following output:

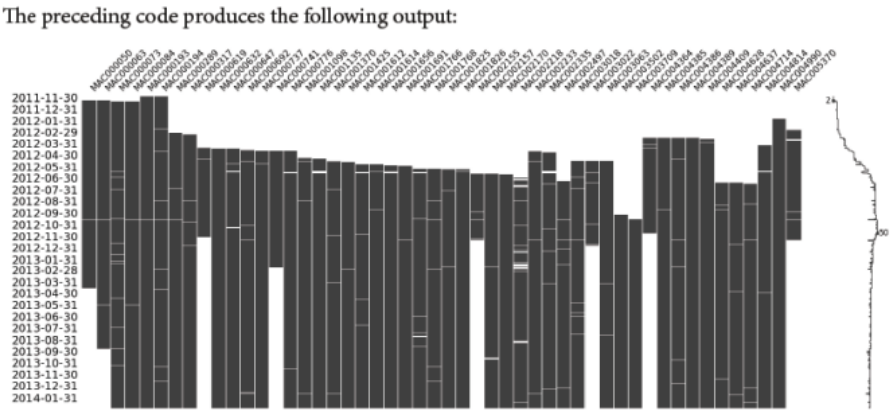

Figure 2.9 – Visualization of the missing data in block 7

The visualization shows dates on the Y-axis and different households on the X-axis. It highlights that not all time series start and end at the same time, as indicated by large white gaps at the beginning of some series (showing data collection started later) and some series finishing earlier than others (indicating they may have stopped collecting data). Smaller white lines represent actual missing values.

Additionally, a sparkline on the right provides a compact view of the number of missing values in each row. If there are no missing values, the sparkline will be far right; if there are many, it will be further left.

It's important to note that just because there are missing values, we shouldn't immediately fill or impute them. The decision to impute missing data comes later in the workflow, depending on the specific needs of different models, as various imputation methods may apply.

So, for now, let's pick one LCLid and dig deeper. We already know that there are some missing values between 2012-09-30 and 2012-10-31. Let's visualize that period:

```
# Taking a single time series from the block
ts_df = exp_block_df[exp_block_df.LCLid == "MAC000193"].set_index("timestamp")
msno.matrix(ts_df["2012-09-30": "2012-10-31"], freq="D")
```

The preceding code produces the following output:

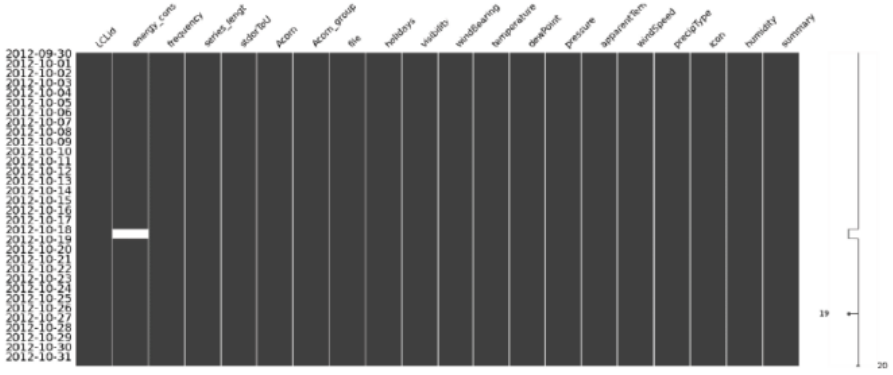

Figure 2.10 – Visualization of missing data of MAC000193 between 2012-09-30 and 2012-10-31

We found that the missing data occurs between October 18 and 19, 2012. Instead of just filling in those gaps, we will create a new section of artificial missing data to see how different methods handle it. We will focus on the dates from October 7 to 8, 2012, and will create a new column with missing values during that time.

```
# The dates between which we are nulling out the time series
window = slice("2012-10-07", "2012-10-08")
# Creating a new column and artificially creating missing values
ts_df['energy_consumption_missing'] = ts_df.energy_consumption
ts_df.loc[window, "energy_consumption_missing"] = np.nan
```

Now, let's plot the missing area in the time series:

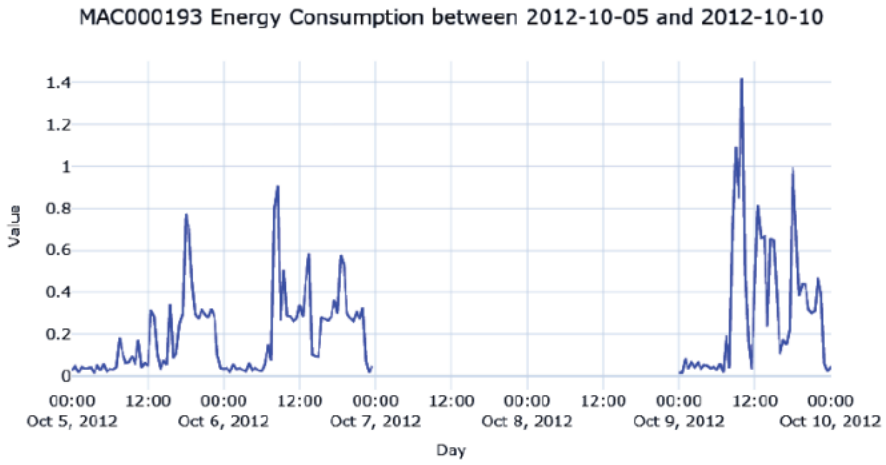

Figure 2.11 – The energy consumption of MAC000193 between 2012-10-05 and 2012-10-10

We are missing energy consumption data for two full days, totaling 96 missing readings (since readings are taken every half hour). If we use simple methods like interpolation to fill these gaps, the result will often look like a straight line, as these methods aren't sophisticated enough to recognize longer-term patterns.

Imputing with the Previous Day

To address these large gaps, one approach is to use the energy readings from the previous day. Since energy usage tends to follow daily patterns, we can fill in the missing data by copying the readings from the same time on the previous day. For example, we can use the reading from 10:00 A.M. on October 17, 2012, to fill in the reading for 10:00 A.M. on October 18, 2012.

```
# Shifting 48 steps to get previous day
ts_df["prev_day"] = ts_df['energy consumption'].shift(48)


# Using the shifted column to fill missing
ts_df['prev_day_imputed'] = ts_df['energy_consumption_missing']

ts_df.loc[null_mask, "prev_day_imputed"] = ts_df.loc[null_mask, "prev_day"]
mae = mean_absolute_error(ts_df.loc[window, "prev_day_imputed"], ts_df.loc[window, "energy consumption"])
```

Let's see what the imputation looks like:

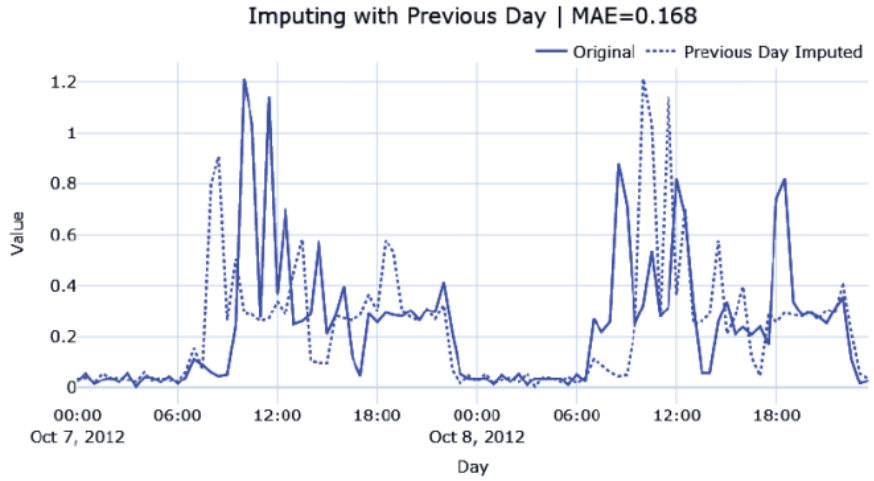

Figure 2.12 – Imputing with the previous day

While this looks better, this method is fragile. Copying the previous day's values assumes that any variations or anomalies are also repeated. We can see that the patterns for the day before and the day after differ.

Hourly Average Profile

A better approach is to calculate an hourly profile, which is the mean consumption for every hour, and use this average to fill in missing data:

```
# Create a column with the Hour from timestamp
ts_df["hour"] = ts_df.index.hour

# Calculate hourly average consumption
hourly_profile = ts_df.groupby(['hour'])['energy_consumption'].mean().reset_index()
hourly_profile.rename(columns={"energy_consumption": "hourly_profile"}, inplace=True)

# Saving the index because it gets lost in merge
idx = ts_df.index

# Merge the hourly profile dataframe to ts dataframe
ts_df = ts_df.merge(hourly_profile, on=['hour'], how='left', validate="many_to_one")
ts_df.index = idx

# Using the hourly profile to fill missing values
ts_df['hourly_profile_imputed'] = ts_df['energy_consumption_missing']
ts_df.loc[null_mask, "hourly_profile_imputed"] = ts_df.loc[null_mask, "hourly_profile"]
```

```
mae = mean_absolute_error(ts_df.loc[window, "hourly_profile_imputed"], ts_df.loc[window, "energy_consumption"
```

Let's see if this is better:

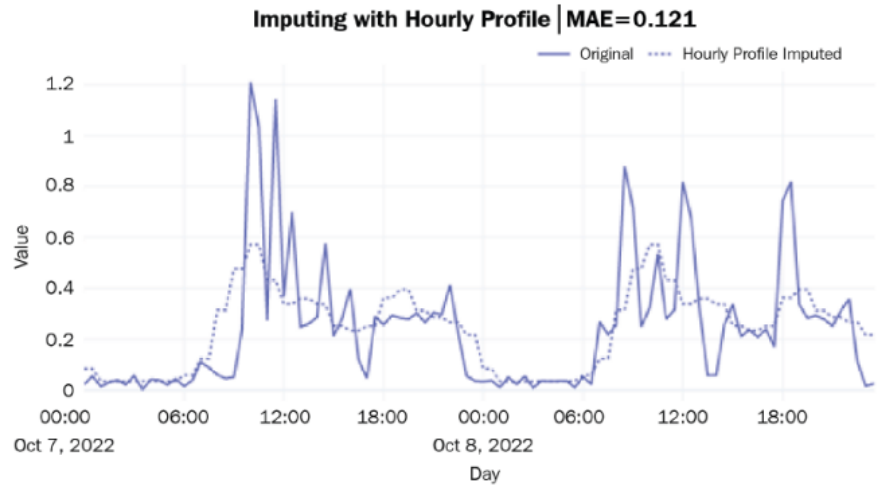

Figure 2.13 – Imputing with an hourly profile

This gives us a much more generalized curve that lacks the spikes observed in individual days. The hourly fluctuations have also been captured as expected, resulting in a lower mean absolute error (MAE).

The Hourly Average for Each Weekday

We can refine this approach by creating a specific profile for each weekday. Usage patterns on weekdays differ from those on weekends, so we calculate average hourly consumption for each weekday:

```
# Create a column with the weekday from timestamp
ts_df["weekday"] = ts_df.index.weekday

# Calculate weekday-hourly average consumption
day_hourly_profile = ts_df.groupby(['weekday', 'hour'])['energy_consumption'].mean().reset_index()
day_hourly_profile.rename(columns={"energy_consumption": "day_hourly_profile"}, inplace=True)

# Saving the index because it gets lost in merge
idx = ts_df.index

# Merge the day-hourly profile dataframe to ts dataframe
ts_df = ts_df.merge(day_hourly_profile, on=['weekday', 'hour'], how='left', validate="many_to_one")
ts_df.index = idx

# Using the day-hourly profile to fill missing values
```

```
ts_df['day_hourly_profile_imputed'] = ts_df['energy_consumption_missing']
ts_df.loc[null_mask, "day_hourly_profile_imputed"] = ts_df.loc[null_mask, "day_hourly_profile"]

mae = mean_absolute_error(ts_df.loc[window, "day_hourly_profile_imputed"], ts_df.loc[window, "energy_consumpt
```

Let's see what this looks like:

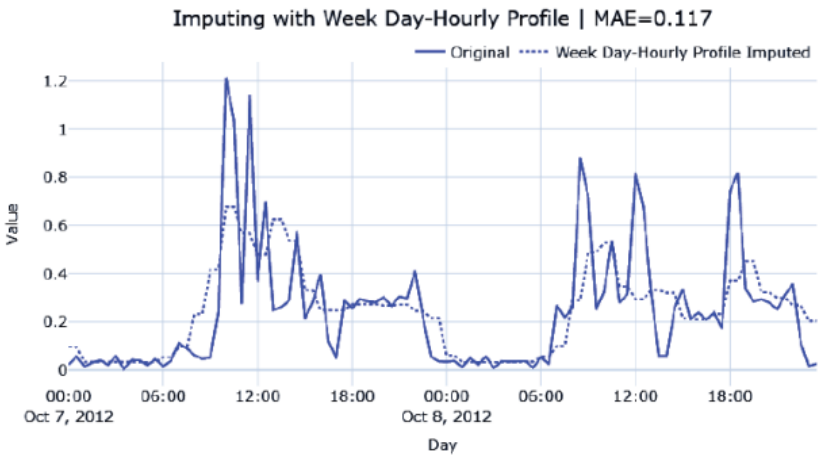

Figure 2.14 – Imputing the hourly average for each weekday

The imputed energy consumption data looks similar to previous weekday profiles because they share similar patterns. The Mean Absolute Error (MAE) is also lower than the MAE from the daily profile.

Seasonal Interpolation

While using seasonal profiles to fill in missing data works well, it can struggle when there is a trend in the data. Simple seasonal profiles don't consider these trends.

To better handle missing data in such cases, we can follow these steps:

1. Calculate the seasonal profile, similar to how we found averages earlier.
2. Subtract the seasonal profile from the original data and use interpolation methods to fill in the missing values.
3. Add the seasonal profile back to the interpolated data.

This method is available in the GitHub repository for this book, specifically in the `src/imputation/interpolation.py` file. We can use it like this:

```
from src.imputation.interpolation import SeasonalInterpolation

# Seasonal interpolation using 48*7 as the seasonal period.
recovered_matrix_seas_interp_weekday_half_hour = SeasonalInterpolation(seasonal_period=48*7, decomposition)
strategy="additive", interpolation_strategy="spline",
interpolation_args={"order":3}, min_value=0).fit_transform(
```

```
    ts_df.energy_consumption_missing.values.reshape(-1, 1))

ts_df['seas_interp_weekday_half_hour_imputed'] = recovered_matrix_seas_interp_weekday_half_hour
```

We are using a method called seasonal interpolation to fill in the missing energy consumption data. The key idea is to find patterns that repeat over a specific period, defined by the parameter `seasonal_period`. In this case, we set `seasonal_period=48` to capture daily patterns, since there are 48 half-hour readings in a day.

We also specify how we want to perform the interpolation, choosing a spline method with a defined order. This method employs seasonal decomposition to separate the seasonal patterns from the data, which will be further discussed in Chapter 3.

We conduct seasonal interpolation using both daily (`48`) and weekly (`48*7`) data, and we plot the results of the imputed data:

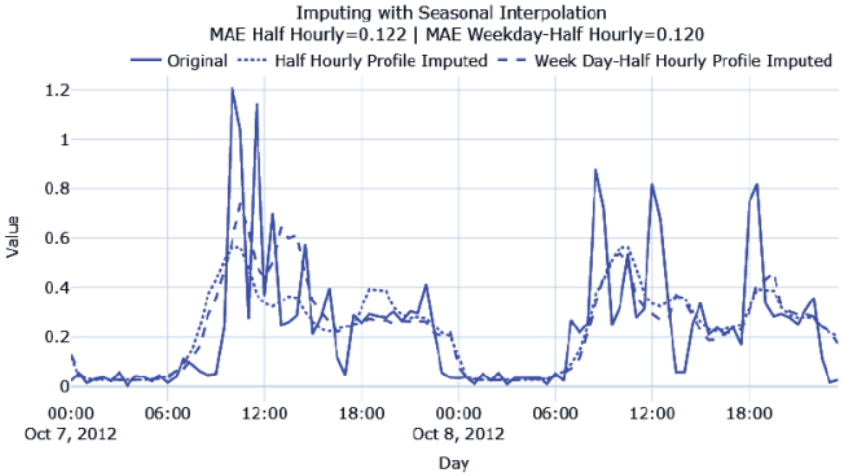

Figure 2.15 – Imputing with seasonal interpolation

Both methods successfully captured seasonal patterns, but the half-hourly profile for each weekday performed better in highlighting peaks on the first day, leading to a lower mean absolute error (MAE). The hourly averages didn't improve much since the time series doesn't show strong trends.

Analyzing and Visualizing Time Series Data

Introduction

In the previous chapter, we learned where to obtain time series datasets, as well as how to manipulate time series data using `pandas`, handle missing values, and so on. Now that we have the processed time series data, it's time to understand the dataset, which data scientists call *Exploratory Data Analysis (EDA)*. EDA is a process by which the data scientist analyzes the data by looking at aggregate statistics, feature distributions, visualizations, and so on to try and uncover patterns in the data that they can leverage in modeling. In this chapter, we will look at a couple of ways to analyze a time series dataset, a few specific techniques that are tailor-made for time series, and review some of the visualization techniques for time series data.

Topics Covered

In this chapter, we will cover the following topics:

- Components of a time series
- Visualizing time series data
- Decomposing a time series
- Detecting and treating outliers

Technical Requirements

You will need to set up the Anaconda environment following the instructions in the Preface of the book to get a working environment with all the packages and datasets required for the code in this book.

Components of a Time Series

Before we start analyzing and visualizing time series, we need to understand the structure of a time series. Any time series can contain some or all of the following components:

- **Trend**
- **Seasonal**
- **Cyclical**
- **Irregular**

These components can be mixed in different ways, but two very commonly assumed ways are *additive* ($Y = \text{Trend} + \text{Seasonal} + \text{Cyclical} + \text{Irregular}$) and *multiplicative* ($Y = \text{Trend} \times \text{Seasonal} \times \text{Cyclical} \times \text{Irregular}$).

The Trend Component

The trend is a long-term change in the mean of a time series. It is the smooth and steady movement of a time series in a particular direction. When the time series moves upward, we say there is an upward or increasing trend, while when it moves downward, we say there is a downward or decreasing trend. At the time of writing, if we think about the revenue of Tesla over the years, as shown in Figure 3.1, we can see that it has been increasing consistently for the last few years.

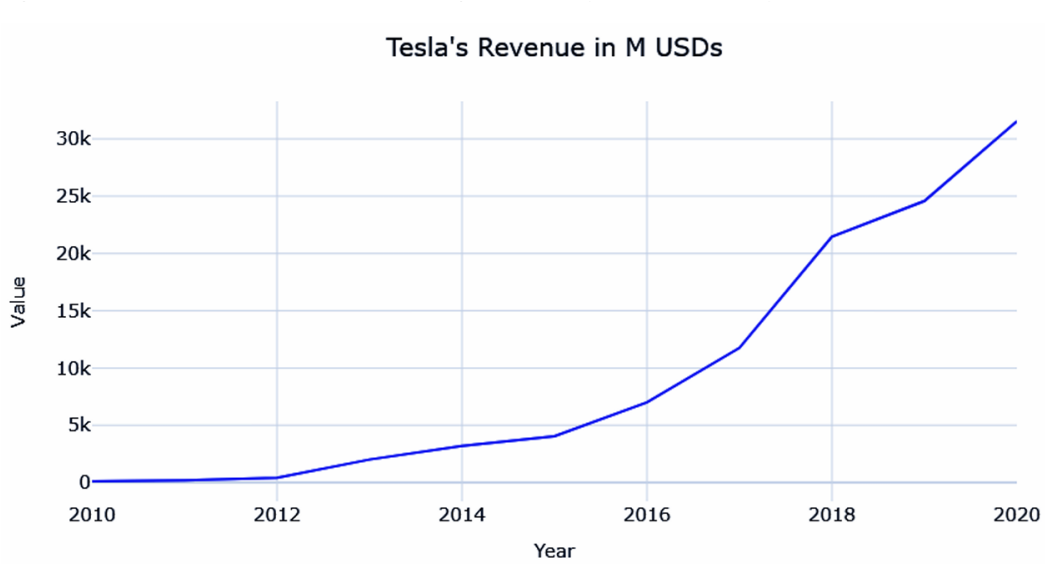

Figure 10: Trend line estimated using LOESS regression.

Looking at the preceding figure, we can say that Tesla's revenue shows an increasing trend. The trend doesn't need to be linear; it can also be non-linear.

The Seasonal Component

When a time series exhibits regular, repetitive, up-and-down fluctuations, we call that seasonality. For instance, retail sales typically increase during the holidays, specifically Christmas in western countries. Similarly, electricity consumption peaks during the summer months in the tropics and the winter months in colder countries. In all these examples, a specific up-and-down pattern repeats every year. Another example is sunspots, as shown in Figure 3.2.

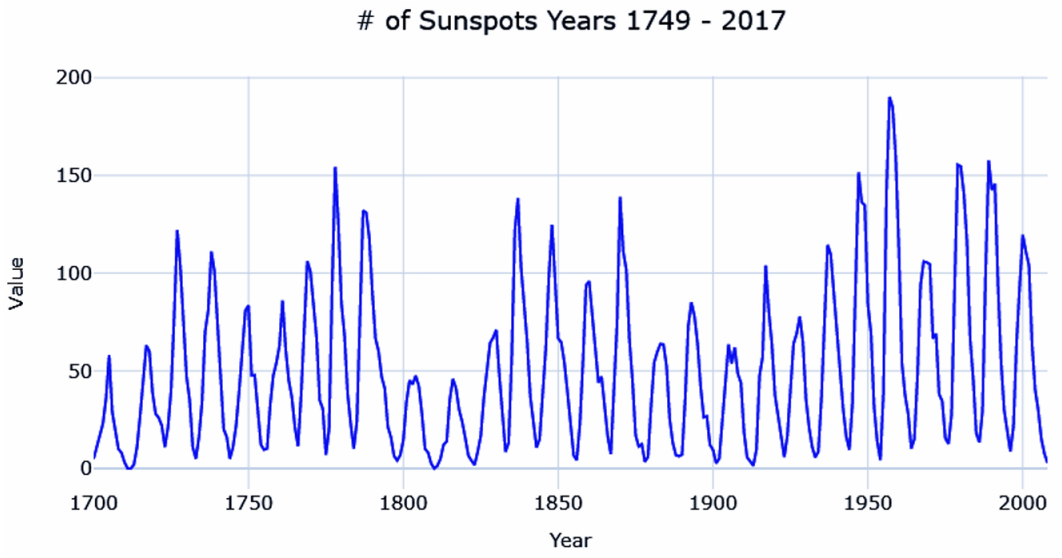

Figure 3.2 – Number of sunspots from 1749 to 2017

Figure 11: Trend line estimated using LOESS regression.

As you can see, sunspots peak every 11 years.

The Cyclical Component

The cyclical component is often confused with seasonality, but it differs due to a subtle difference. Like seasonality, the cyclical component also exhibits a similar up-and-down pattern around the trend line, but instead of repeating the pattern every period, the cyclical component is irregular. A good example of this is economic recession, which happens over a 10-year cycle. However, this doesn't occur on a strict schedule; sometimes, it can be fewer or more than every 10 years.

The Irregular Component

This component remains after removing the trends, seasonality, and cyclicity from a time series. Traditionally, this component is considered unpredictable and is also called the *residual* or *error term*. In classical statistics-based models, the goal is to capture all other components so that the only remaining part is the irregular component. In modern machine learning, we do not consider this component entirely unpredictable. We try to capture this component, or parts of it, by using exogenous variables. For instance, the irregular component of retail sales may be explained by various promotional activities. With this additional information, the "unpredictable" component becomes predictable again. However, no matter how many additional variables you add to the model, there will always be some true irregular component (or true error) left behind.

Now that we know the different components of a time series, let's explore how to visualize them.

Visualizing time series data

In Chapter 2, we learned how to prepare a data model for analyzing new datasets. Exploratory Data Analysis (EDA) is the next step, aimed at understanding the dataset, identifying patterns, spotting anomalies, and forming hypotheses. EDA helps in selecting features and modeling techniques.

Line Charts

A line chart plots time on the X-axis and values on the Y-axis, ideal for initial time series insights. For long, high-variation series, a rolling average (e.g., monthly) provides a smoother view.

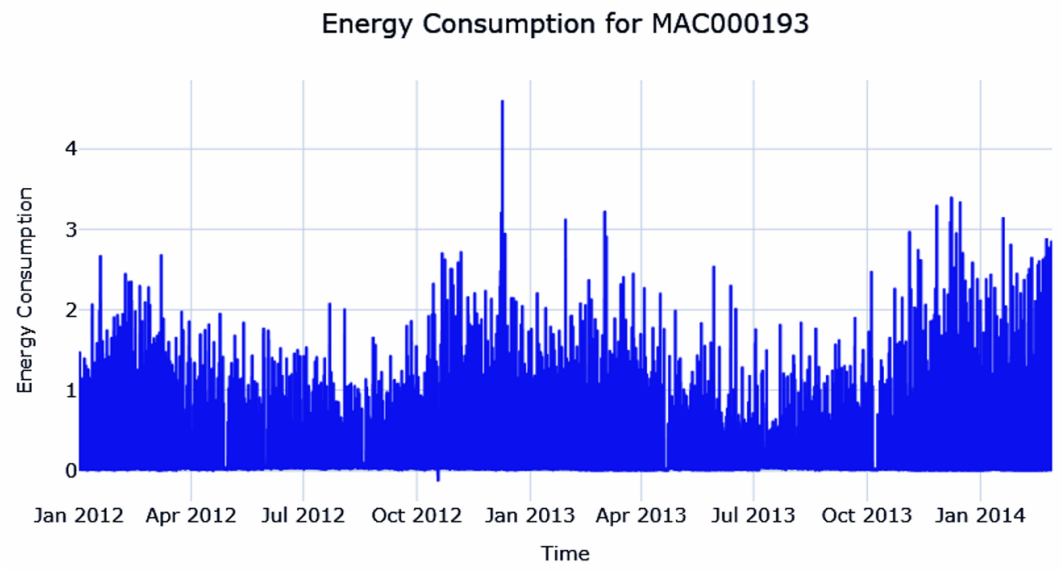

Figure 12: Line plot of household MAC000193

Seasonal Plots

Seasonal plots show recurring patterns by representing cycles in different colors or line styles. For example, plotting monthly energy consumption across years reveals consistent seasonal trends.

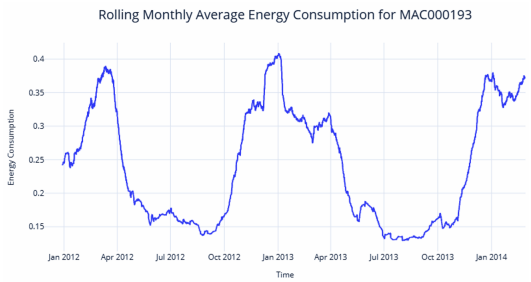

Figure 13: Seasonal plot at monthly resolution

Seasonal Box Plots

Seasonal box plots reduce clutter, showing data variability across cycles. The median, interquartile range, and outliers provide insights into seasonal changes.

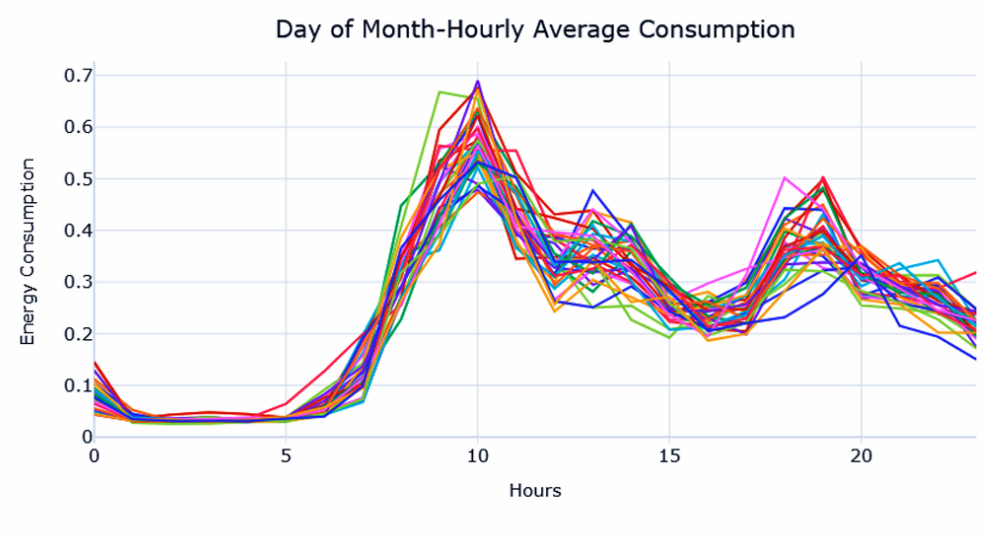

Figure 14: Seasonal plot at monthly resolution

Calendar Heatmaps

Calendar heatmaps display data in two time dimensions (e.g., weekdays and hours) with color intensity indicating values, offering a condensed view of patterns.

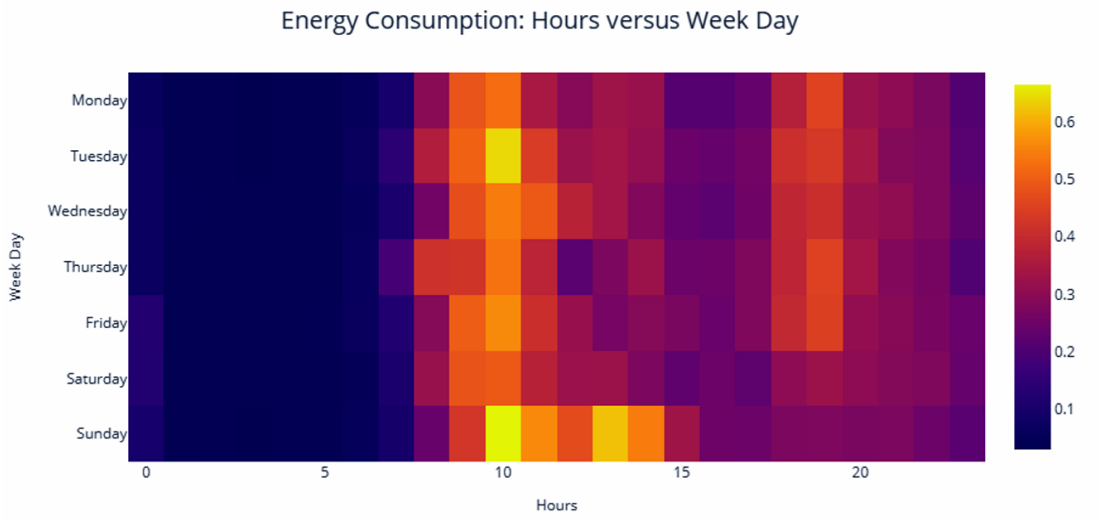

Figure 15: Seasonal plot at monthly resolution

Autocorrelation Plots

Autocorrelation plots show correlations between time series values across different lags, useful for understanding dependencies, particularly in forecasting.

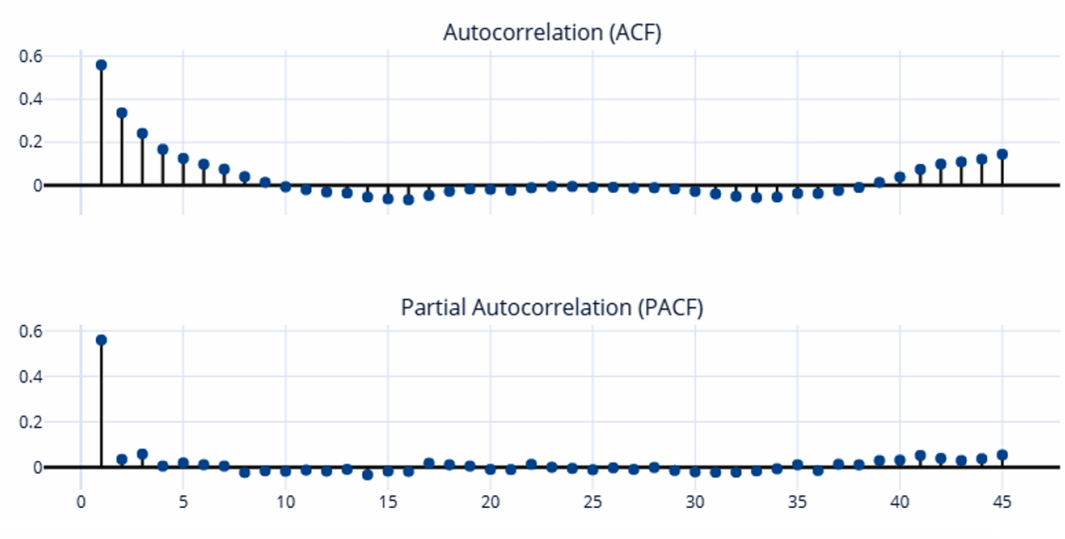

Figure 16: Seasonal plot at monthly resolution

Best Practice

To make a time series stationary, a quick approach is to use seasonal decomposition and analyze the residuals, which are typically trend- and seasonality-free.

Decomposing a Time Series

Seasonal decomposition is the process by which we deconstruct a time series into its components—typically, trend, seasonality, and residuals. The general approach for decomposing a time series is as follows:

1. **Detrending:** Estimate the trend component (the smooth change in the time series) and remove it from the time series, giving us a detrended time series.
2. **Deseasonalizing:** Estimate the seasonality component from the detrended time series. After removing the seasonal component, the remaining data is the residual.

Detrending

Detrending can be performed in a few different ways. Two popular methods include:

Moving Averages

One of the simplest methods for estimating trends is by applying a moving average along the time series. This involves a window that slides along the series, recording the average of all values within the window

at each step. Although this results in a smoothed-out time series and helps estimate the slow change in the series (the trend), it can be noisy. Ideally, noise should reside in the residuals rather than the trend.

LOESS

The LOESS algorithm (locally weighted polynomial regression), developed by Bill Cleveland from the 1970s to the 1990s, is a non-parametric method for fitting a smooth curve to a noisy signal. It estimates a smoothed trend by weighting nearby points more heavily. This provides a robust way to model smooth changes in the time series (trend).

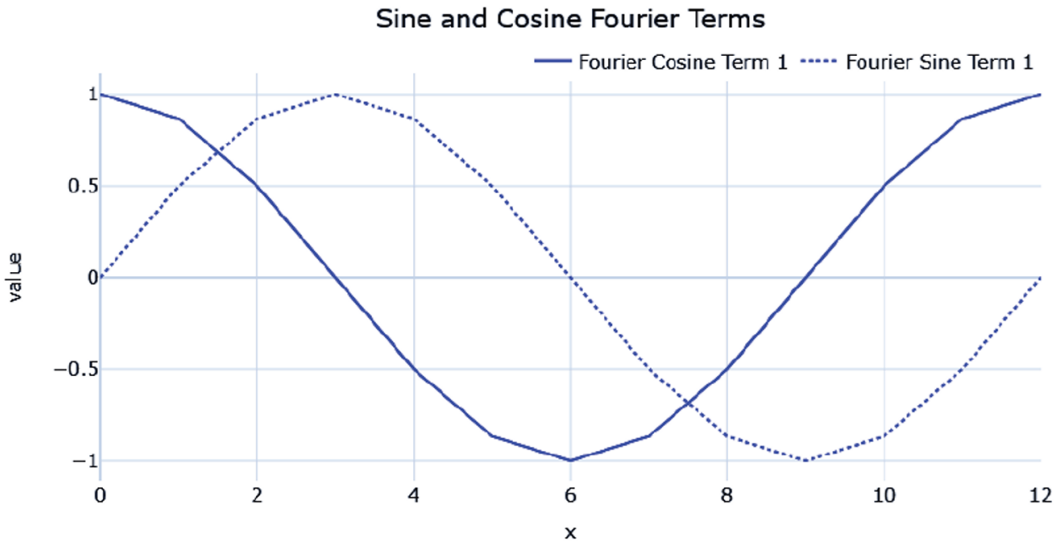

Figure 17: Trend line estimated using LOESS regression.

Deseasonalizing

The seasonality component can also be estimated in various ways, including:

Period-Adjusted Averages

For a time series with a repeating cycle, we can calculate a seasonality index by averaging values over corresponding periods in each cycle. For instance, in a monthly time series with annual seasonality, we take the average of all January values to get a seasonality index for January, and so forth. This results in period averages that can be transformed into a seasonality index.

Fourier Series

Joseph Fourier's theory allows any periodic function to be approximated by sine and cosine waves. This characteristic helps in extracting seasonality from a time series, as seasonality itself is periodic. The sine-cosine form of a Fourier series for the N-term approximation of the signal $S(x)$ is:

$$S(x) = \frac{a_0}{2} + \sum_{n=1}^{N} \left(a_n \cos\left(\frac{2\pi n x}{P}\right) + b_n \sin\left(\frac{2\pi n x}{P}\right) \right)$$

where P is the length of the cycle, and a_n and b_n are Fourier coefficients.

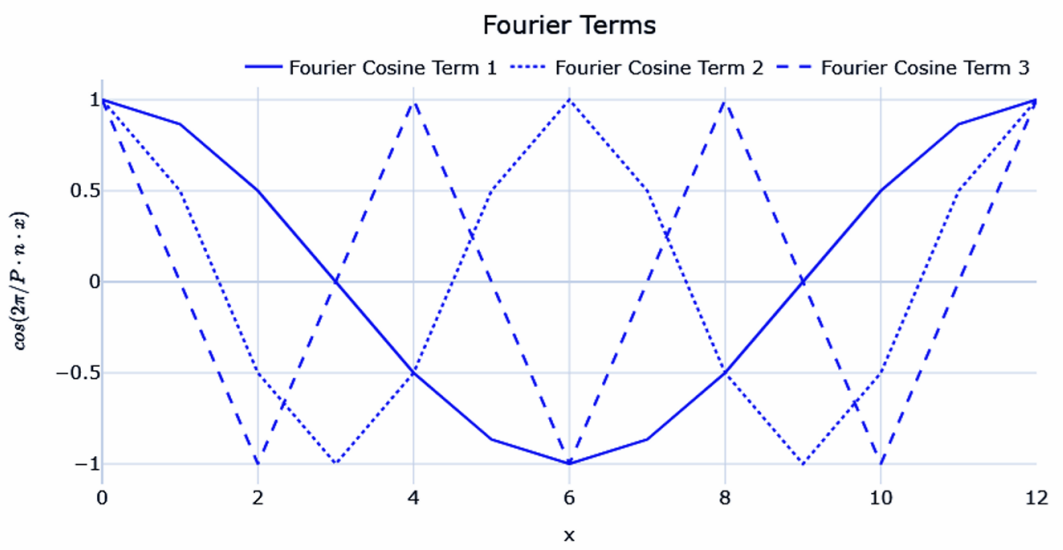

Figure 18: Fourier series terms for seasonal decomposition.

Key Parameters for Seasonal Decomposition

- **period**: Defines the seasonal period expected in the data.
- **model**: Determines the type of decomposition (additive or multiplicative).
- **filt**: Specifies weights in the moving average for smoothing the trend.
- **extrapolate_trend**: Extends the trend component to avoid missing values.
- **two_sided**: Controls whether the moving average uses past and future values (True) or just past values (False).

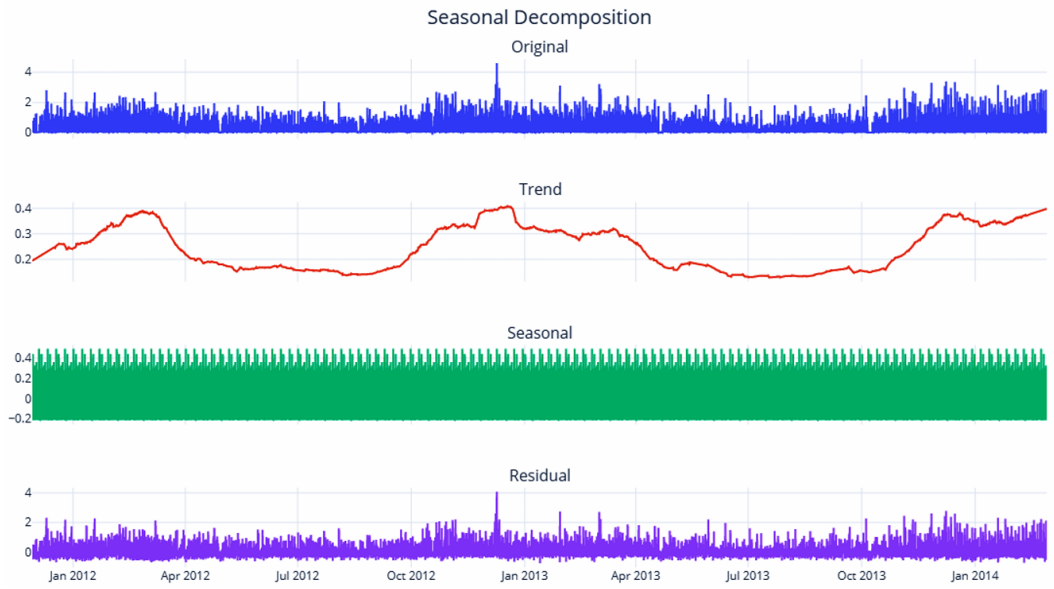

Figure 19: Seasonal decomposition using statsmodels.

Zoomed-In Decomposition

Using Fourier decomposition with custom parameters, we can achieve detailed seasonal component extraction. For example, applying $n_{fourier\_terms}$ allows us to control the complexity of the seasonality extracted from data.

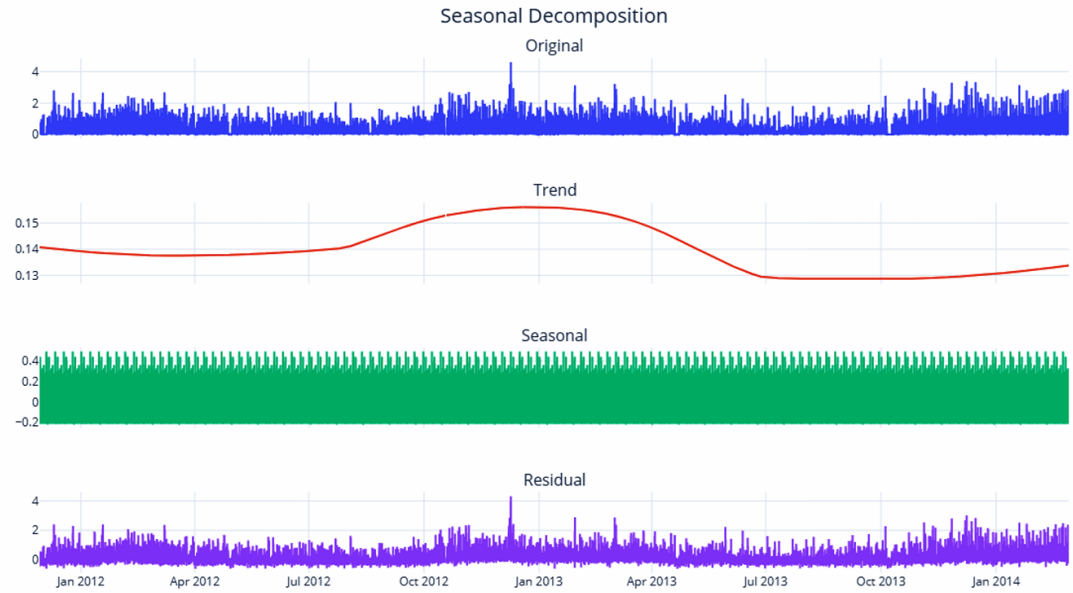

Figure 20: STL decomposition zoomed-in for clearer seasonality patterns.

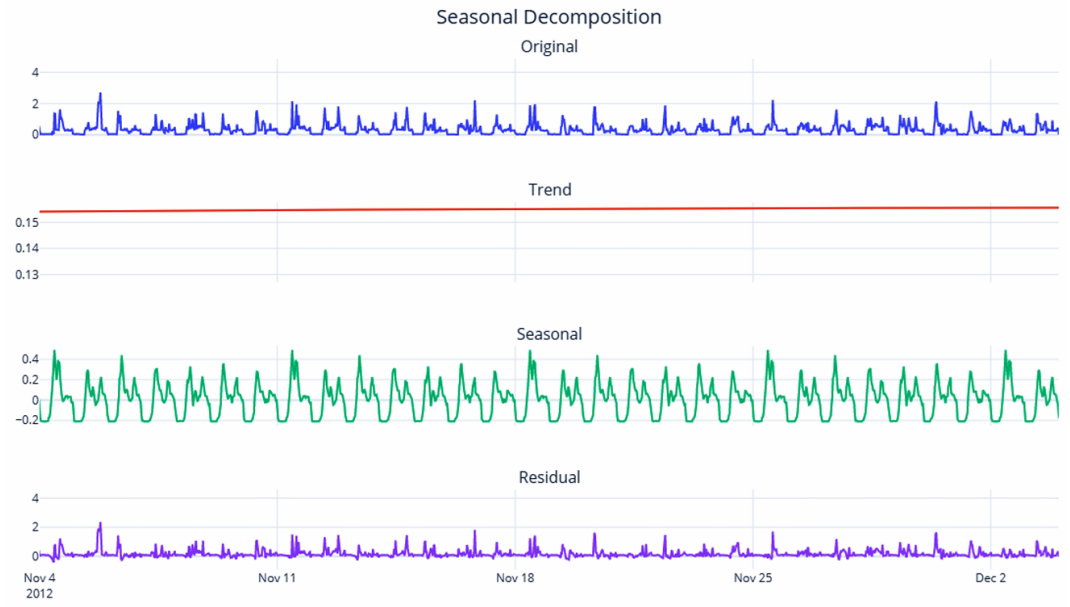

Figure 21: Fourier decomposition (zoomed in for a month).

Detecting and Treating Outliers

An outlier is an observation that lies at an abnormal distance from the rest of the observations. If we consider the data-generating process (DGP) as a stochastic process, outliers are points with the least probability of being generated by the DGP. Causes of outliers include faulty measurement, incorrect data entry, or rare black-swan events. Detecting and treating outliers can help a forecasting model better understand the data.

Outlier detection is a specialized field in time series analysis, but here we cover basic techniques to identify and handle outliers, aiming to improve data quality for forecasting models. For advanced studies, refer to the Further Reading section.

Standard Deviation Method

A commonly used rule of thumb is that if μ is the mean of the time series and σ is the standard deviation, then anything outside the bounds $\mu \pm 3\sigma$ is considered an outlier. This rule is based on the properties of the normal distribution, where 68%, 95%, and 99% of data lie within one, two, and three standard deviations, respectively.

However, this threshold is not strict; adjusting it can help in practical scenarios. For highly seasonal data, deseasonalize the data first, then apply this rule to residuals to avoid marking seasonal peaks as outliers.

Interquartile Range (IQR) Method

The IQR method defines outliers based on quartiles. The upper and lower bounds are defined as:

$$\text{Upper Bound} = Q3 + n \times \text{IQR},$$

$$\text{Lower Bound} = Q1 - n \times \text{IQR},$$

where $\text{IQR} = Q3 - Q1$, and n is the multiple used to set acceptable bounds. For highly variable datasets, IQR is more robust than standard deviation because it is less sensitive to extreme values. Typically, setting $n = 1.5$ achieves similar results to the $\pm 3\sigma$ rule.

Isolation Forest

Isolation Forest is an unsupervised anomaly detection method that isolates outliers by creating a forest of decision trees. Outliers are more likely to fall into shorter branches, while normal points require longer paths. The "anomaly score" is based on the depth at which each point is isolated. The 'contamination' parameter in the scikit-learn 'IsolationForest' implementation specifies the expected proportion of anomalies (0 to 0.5).

Extreme Studentized Deviate (ESD) and Seasonal ESD (S-ESD)

ESD is a statistical technique that iteratively applies Grubbs's test to detect outliers, assuming normality. In 2017, Hochenbaum et al. from Twitter proposed using a generalized ESD with deseasonalization to detect outliers in time series data. This method only requires an upper bound for expected outliers, making it simpler to automate.

	# of Outliers	% of Outliers
3SD	802	2.12%
2SD on Residuals	728	1.92%
4IQR	747	1.97%
4SD on Residuals	468	1.24%
Isolation Forest	364	0.96%
Isolation Forest on Residuals	359	0.95%
ESD	420	1.11%
S-ESD	424	1.12%

Figure 22: Figure 3.19 – Outliers detected using different techniques

Treating Outliers

Once outliers are identified, decide whether to correct them. Automated detection should be reviewed manually to avoid removing valuable patterns. When handling large datasets, automated techniques like replacing outliers with the maximum, minimum, or a percentile value can be used. Another approach is treating outliers as missing values and imputing them.

Modern forecasting models may not require outlier correction. Experimentation can help determine if outlier treatment benefits the model.

Summary

In this chapter, we introduced key components of time series data and techniques for visualization and decomposition. We discussed multiple methods to detect outliers and concluded with strategies for treating them.

Data Pre-processing

Data pre-processing is a crucial step in machine learning and data analysis, impacting model performance significantly. This summary covers general preparation methods and insights from Manu Joseph's Modern Time Series Forecasting with Python, focused on time series data.

Data Cleaning

Removing Missing Values:

Missing values are common in datasets and can hinder model training. Suppose we have daily temperature readings, but some days are missing. Here's how we might handle these gaps with approaches include:

Forward Fill (Last Observation Carried Forward):

Using the last observed value to fill in missing one.For example, If March 2 has a missing temperature but March 1 has a reading of 72°F, forward fill would set March 2's temperature to 72°F.

Backward Fill (Next Observation Carried Backward):

Filling in missing values by carrying backward the next observed value.Here, if March 4 is missing but March 5 is 74°F, backward fill would set March 4 to 74°F.

Mean Value Fill:

Filling missing entries with the mean of the series.Here,If the average temperature in the dataset is 73°F, this value could replace any missing entries.

These techniques are useful when missing data is limited. For more extensive gaps, imputation techniques or predictive models may be necessary.

Outlier Detection and Removal:

The model's comprehension of data patterns may be distorted by outliers. Commonly employed methods include interquartile range (IQR) and Z-score analysis.

For example, A retail dataset may contain daily sales data that primarily falls between the range of 100 to 300 units sold. 10,000 copies sold in a single day could be an anomaly that skews model forecasts. This entry can be flagged using techniques like Z-score or boxplots, allowing you to review or modify it.

Noise Reduction:

Model clarity is increased when noise (irrelevant data) is reduced. Depending on the data, either filtering or smoothing techniques can be used to achieve this.

For example, Moving averages can be used to smooth out short spikes caused by motion artifacts in a time series containing hourly heart rate data. A cleaner signal for additional investigation is obtained by averaging observations over brief periods of time (e.g., every five hours).

Data Transformation

Normalization and Standardization:

For uniformity across features, data must be scaled to a certain range (for example, 0–1 for normalization) or adjusted to have a mean of 0 and a standard deviation of 1 (standardization). For models like neural networks that are sensitive to the size of the data, these methods are crucial. For example, In order to align feature values for balanced model weighting, normalization scales features to 0–1, and standardization modifies them to a mean of 0 and standard deviation of 1.

Encoding Categorical Variables:

Converting numerical data from categorical data,For example, If we have data on customer feedback with categories like "Satisfied," "Neutral," and "Dissatisfied": typically by using:

1. **One-Hot Encoding:** Creates binary columns for each category like Satisfied=1, Neutral=0, Dissatisfied=0.
2. **Label Encoding:** Assigns each category a unique integer e.g., Satisfied = 2, Neutral = 1, Dissatisfied = 0..

Feature Scaling:

Assists in model convergence by ensuring that numerical features have comparable ranges.

Feature Engineering for Time Series:

To capture temporal patterns in time series, this entails developing rolling averages, seasonal indicators, and lagged variables. For example, Lagged features (such as previous day/week sales) and rolling averages (such as the 7-day moving average) provide context for historical trends and seasonal patterns in a sales dataset.

Data Formatting for Time Series

Compact vs. Expanded Form: In time series, formatting can vary:

1. Compact Form:

It is memory-efficient and perfect for datasets with regular intervals because each time series takes up one row in a DataFrame, with timestamps as columns.

For example, Hourly readings would be represented as columns (e.g., Hour 1, Hour 2, etc.), with each day being a row.

2. Expanded Form:

It is appropriate for time series with varied intervals since each observation takes up a distinct row, clearly capturing each time step.

For example, Perfect for managing irregular intervals or combining disparate data sources, each hour is represented as a row with Date, Time, and Energy Usage columns.

Dimensionality Reduction

Principal Component Analysis (PCA) and Linear Discriminant Analysis (LDA):

LDA maximizes class separability, while PCA reduces dimensionality while maintaining data variance. These are particularly useful for datasets with several dimensions.

For example, With 100 daily health metrics (e.g., heart rate, steps, sleep quality), PCA can reduce these to a few principal components, retaining key variability and simplifying analysis.

Feature Selection:

The most pertinent features are chosen using methods like Recursive Feature Elimination (RFE), which lowers model complexity and improves interpretability.

For example, RFE simplifies models and enhances interpretability by emphasizing important information in a housing dataset, such as bedroom count, neighborhood rating, and square footage.

Data Splitting

Train-Test Split:

Separates data into subsets for testing and training in order to assess the model. Keeping time series in chronological sequence is essential.

In an email open-rate dataset, using the first 70% for training and the last 30% for testing preserves chronological order, crucial for time series accuracy.

Cross-Validation for Time Series (Backtesting):

By training and verifying in segments, frequently with a rolling or expanding window technique, it guarantees that the model generalizes effectively.

Time series cross-validation helps generalize the model by simulating real forecasting through the use of a rolling window (e.g., train on Jan–Mar, test on Apr; then Feb–Apr, test on May).

Handling Imbalanced Data

A model may be biased toward more frequent classes if the classes are unbalanced. This problem can be resolved with the use of strategies like class weighting, SMOTE (Synthetic Minority Over-sampling Technique), and resampling.

The dataset is balanced by using SMOTE to artificially create more infrequent system failures in log data, which increases the model's sensitivity to failures.

For example, When monthly website traffic is broken down into trend, seasonal, and residual components, feature engineering insights are obtained, such as the ability to capture monthly seasonality.

Exploratory Data Analysis (EDA) and Visualization for Time Series

For time series, EDA entails breaking the series down into its trend, seasonal, cyclic, and irregular components. Line plots, rolling averages, and seasonal decomposition are examples of visualization techniques that assist uncover trends, abnormalities, and insights that guide feature engineering and model selection.

For instance, Plotting a rolling average for daily sales smoothes volatility and aids in identifying trends and anomalies during EDA. Line plots are a useful way to visualize time series data.

Data Preparation

The following resources offer thorough treatment of data preparation for more in-depth understanding:

Data Preparation for Data Mining" by Dorian Pyle

For those who are unfamiliar with data preparation, this book provides a comprehensive introduction to the fundamental ideas of data preprocessing, from feature engineering to cleaning.

Data Science for Business" by Foster Provost and Tom Fawcett

Useful advice for preparing data in commercial settings.

Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow" by Aurélien Géron

Discusses preparation methods with well-known Python libraries.

Mathematical handling of missing data

The book Modern Time Series Forecasting with Python covers essential techniques for handling missing data in time series, crucial for model accuracy and predictive power, with examples and illustrations.

Simple Imputation Techniques

These techniques use simple techniques to use the data that is already available to fill in the missing numbers.

1. Forward Fill (Last Observation Carried Forward):

Fills missing values with the last observed entry, useful for intermittent gaps where prior values offer a reasonable estimate.

2. Backward Fill (Next Observation Carried Backward):

Similar to forward fill, but fills missing values with the next observed value. Useful in cases where future values are more representative of the missing period.

3. Mean/Median Imputation:

Replaces missing values with the mean or median of the available data, providing a constant imputation that works best when data lacks trend or seasonality. In python, this can be done using:

```
df['value_column'].ffill()  # Forward fill
df['value_column'].bfill()  # Backward fill
df['value_column'].fillna(df['value_column'].mean())  # Mean fill
```

For illustration, we can analyze PM2.5 readings from the Monash region, which includes some missing values. This dataset is published by the ACT Government of Canberra, Australia, under the CC BY Attribution 4.0 International License:
https://www.data.act.gov.au/Environment/Air-Quality-Monitoring-Data/94a5-zqnn.

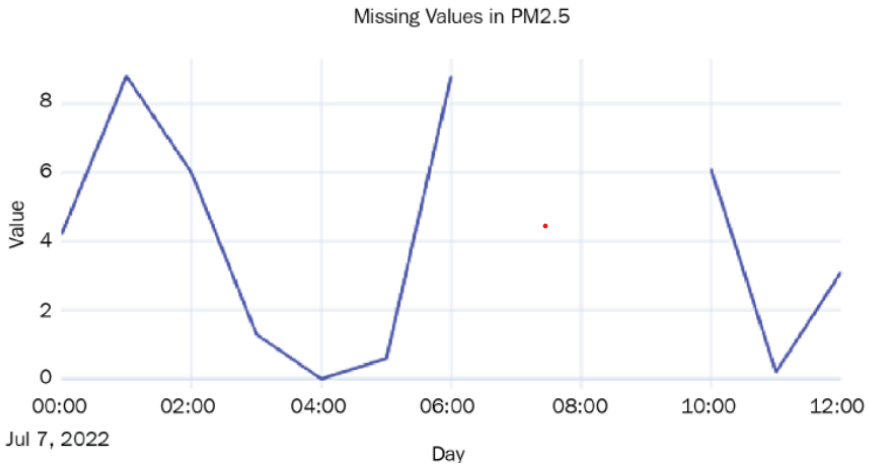

Figure 23: Missing values in the Air Quality dataset

Let's plot the imputed lines we get from using these three techniques:

Interpolation Methods

Interpolation, which can be especially useful for regularly spaced time series data, estimates missing values by examining trends or patterns between observed values.

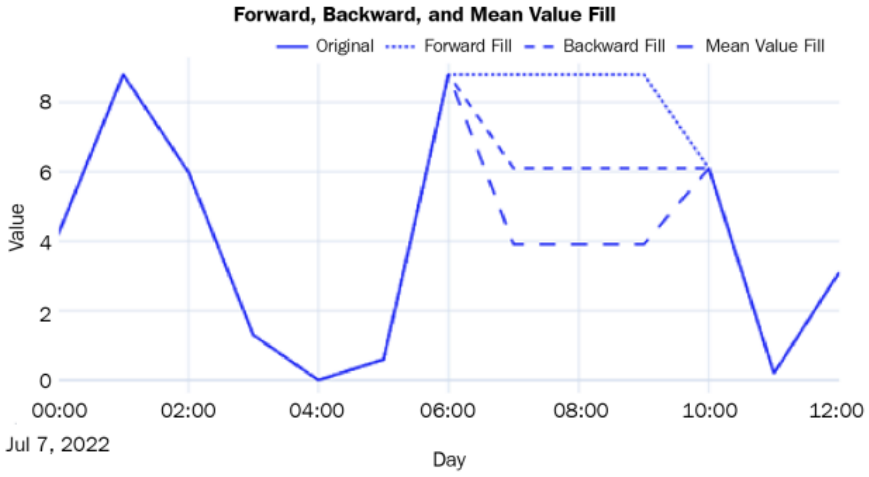

Figure 24: Imputed lines using Forward Fill, Backward Fill, and Mean Value Fill

1. Linear Interpolation:

Fills in the missing values based on the assumption that the data points follow a linear trend.Example code for linear interpolation:

```
df['value_column'].interpolate(method="linear")
```

2. Polynomial Interpolation:

For data with a curve, polynomial interpolation can help. It fits a polynomial (like a quadratic or cubic curve) to the data, filling in gaps in a way that follows the overall shape of the series.

3. Spline Interpolation:

This is a bit more flexible, using a series of polynomial segments (splines) to create a smooth curve that connects your data points. It's ideal if the data has complex curves or undulating trends.

Let's plot these two non-linear interpolation techniques:

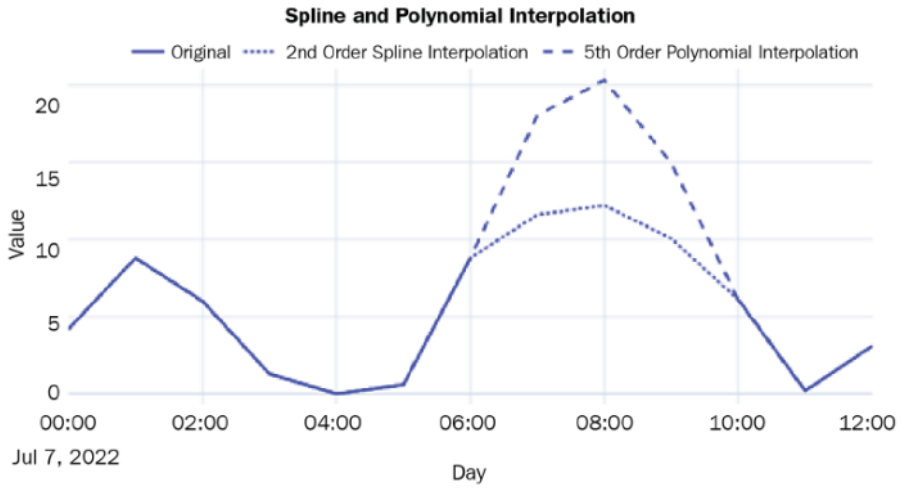

Figure 25: Spline and polynomial Interpolation

Advanced Statistical and Machine Learning Approaches

For more complex data, statistical models and machine learning can predict missing values by considering the overall patterns and relationships within the data.

Kalman Filtering:

Often used in time series, this method can estimate missing values even in noisy or trending data. It works by updating predictions at each step, accounting for both data trends and any random variations. For example, using Kalman filtering in Python:

```
from pykalman import KalmanFilter
kf = KalmanFilter(initial_state_mean=0, n_dim_obs=1)
smoothed_data, _ = kf.smooth(df['value_column'])
```

Expectation-Maximization (EM):

This iterative approach fills in missing values by assuming a likely data distribution (like a normal curve). It makes an initial guess, then refines it with each iteration to improve accuracy.

Machine Learning Models (e.g., K-Nearest Neighbors, Random Forests):

These models can predict missing points based on patterns in the data. For example, Random Forests may use other columns or variables in a dataset to estimate missing entries, which is helpful for multivariate time series data.

Visualizing Imputation Methods

Visual comparisons aid in selecting the best imputation method by showing how well each maintains data patterns. Plotting original vs. imputed values (e.g., linear, spline, forward-fill) reveals differences—spline often yields smoother trends, while forward fill may appear flatter.

We have to choose a method that suits our data's nature: interpolation works for steady trends, while machine learning is better for unpredictable patterns.

The goal is a complete, accurate dataset that reflects underlying trends, balancing simplicity with accuracy.

Reference

- Zhang, A., & Zheng, Y. (2020). Multivariate Time Series Forecasting with Missing Values. Focuses on handling incomplete multivariate time series data using imputation techniques.
- Joseph, Manu. *Modern Time Series Forecasting with Python: Explore Industry-Ready Time Series Forecasting Using Modern Machine Learning and Deep Learning.* Packt Publishing, Limited, 2022.

- Kasun Bandara, Rob J. Hyndman, and Christoph Bergmeir. (2021). MSTL: A Seasonal-Trend Decomposition Algorithm for Time Series with Multiple Seasonal Patterns. *arXiv:2107.13462*. https://arxiv.org/abs/2107.13462.
- Hochenbaum, J., Vallis, O., & Kejariwal, A. (2017). Automatic Anomaly Detection in the Cloud Via Statistical Learning. *ArXiv:1704.07706*. https://arxiv.org/abs/1704.07706.
- D. Pyle, Data preparation for data mining. morgan kaufmann, 1999.
- F. Provost and T. Fawcett, "Data science for business."
- A. Géron, Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow. " O'Reilly Media, Inc.", 2022.
- GitHub: https://github.com/AaliyaJaved/TimeSeriesAnalysis_Group1.git