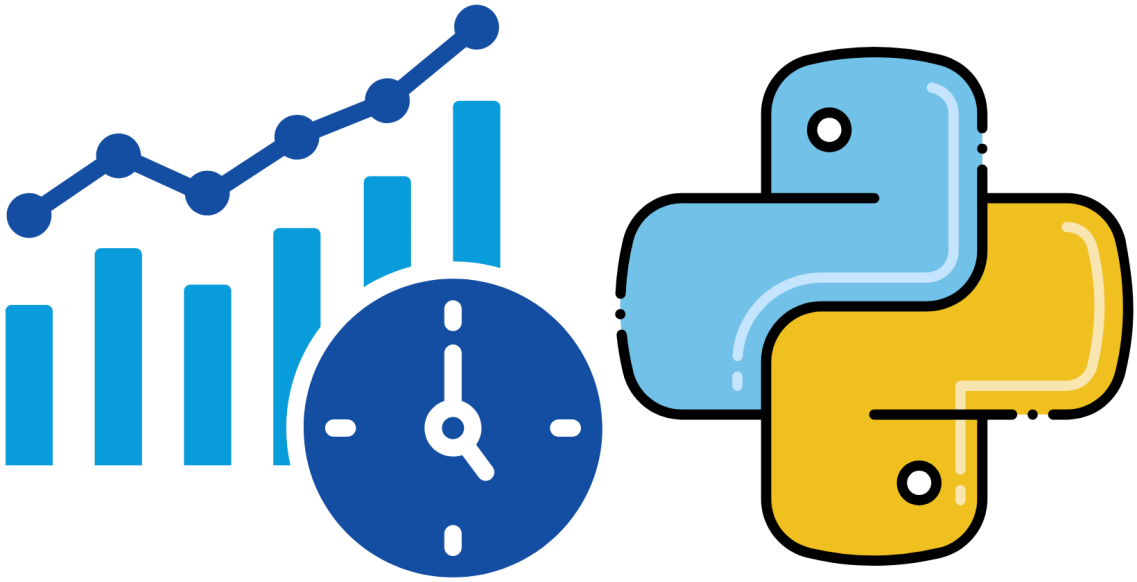
Hands-On Time Series Analysis with Python



To Data & Beyond

Youssef Hosni

Hands-On Time Series Analysis with Python

Hands-On Time Series Analysis with Python

Youssef Hosni

Table of Contents

About this book

Time is the one dimension that distinguishes time series analysis from every other branch of data science. While standard datasets represent a static snapshot of the world, time series data captures the world in motion. From the fluctuating heartbeat of the stock market to the rising trajectory of global temperatures, understanding how to manipulate, analyze, and forecast time-dependent data is a critical skill for the modern data scientist.

Hands-On Time Series Analysis with Python is written to be exactly what the title suggests: a practical, code-first guide. It is designed to bridge the gap between theoretical statistics and the actual implementation of forecasting models using the Python ecosystem. We move quickly past the abstract theory and dive straight into the tools you will use daily: Pandas, Matplotlib, Statsmodels, and Scikit-Learn.

What You Will Learn

This book is structured to take you through the complete lifecycle of a time series project:

- **Chapter 1: Manipulating Time Series Data.** Before you can model, you must master the index. We begin by leveraging the power of Pandas to slice, dice, resample, and window your data, turning messy timestamps into structured financial metrics.
- **Chapter 2: Time Series Data Analysis in Python.** We uncover the statistical properties that drive time series behaviors. You will learn about correlation, autocorrelation, and the crucial concept of stationarity, setting the mathematical foundation for the models to come.
- **Chapter 3: Visualizing Time Series Data.** A picture is worth a thousand timestamps. We explore how to create professional visualizations that decompose complex signals into trends, seasonality, and noise, allowing you to "see" the story behind the data.
- **Chapter 4: Time Series Forecasting with ARIMA Models.** We dive into the gold standard of statistical forecasting. You will master the Box-Jenkins method to build, validate, and automate ARIMA and SARIMA models, capable of capturing complex seasonal patterns.
- **Chapter 5: Time Series Forecasting with Machine Learning.** We cross the bridge into the modern era of predictive analytics. You will learn to treat time series forecasting as a supervised learning problem, mastering feature engineering and rigorous cross-validation techniques to build robust regression models.

Who This Book Is For

This book is intended for data analysts, data scientists, and developers who are comfortable with the basics of Python and want to specialize in time series analysis. Whether you are trying to predict sales for the next quarter, analyze financial volatility, or monitor sensor data, this book provides the blueprint you need to build reliable, predictive solutions.

About the code

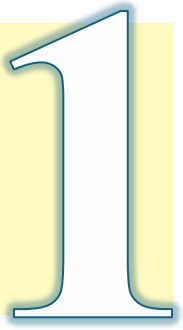
To make it easy to follow up with the book, all the code examples and datasets used in this book can be found in this **GitHub repository.**

About the Author

Youssef Hosni is a data scientist and machine learning researcher who has been working in machine learning and AI for more than seven years. In addition to being a researcher and data science practitioner, Youssef has a strong passion for education. He is known for his leading data science and AI blog, newsletter, and eBooks on data science and machine learning. Youssef is an Applied scientist at Greenstep, focusing on building Generative AI features. He is also an AI applied researcher at Aalto University, working on AI agents and their applications. Before that, he worked as a researcher which he applying deep learning and computer vision techniques to medical images.

Manipulating Time Series Data

Table of Contents:



1. Working with Time Series in Pandas

This section lays the foundations to leverage the powerful time-series functionality made available by how Pandas represents dates, in particular by the DateTimeIndex. You will learn how to create and manipulate date information and time series, and how to do calculations with time-aware DataFrames to shift your data in time or create period-specific returns.

1.1. How to use data and times with pandas

The basic building block of creating a time series data in Python using Pandas time stamp **(pd.Timestamp)** is shown in the example below:

```
import pandas as pd
from datetime import datetime # To manually create dates
time_stamp = pd.Timestamp(datetime(2017, 1, 1))
```

The timestamp object has many attributes that can be used to retrieve specific time information of your data, such as year and weekday. In the example below, the year of the data is retrieved.

```
time_stamp.year
```

The second building block is the **period** object. The period object has a freq attribute to store the frequency information. The default is monthly frequency, and you can convert from one frequency to another as shown in the example below.

```
period = pd.Period('2017-01')
period
```

```
Period('2017-01', 'M')
```

The output shows that the default frequency is monthly. You can convert it into a daily frequency using the code below.

```
period.asfreq('D') # convert to daily
```

```
Period('2017-01-31', 'D')
```

You can also convert a period to a timestamp and vice versa. This is shown in the example below.

```
period.to_timestamp().to_period('M')
```

You can do basic data arithmetic operations, for example, starting with a period object for January 2017 at a monthly frequency, just add the number 2 to get a monthly period for March 2017. This is shown in the example below.

```
period = pd.Period('2017-01')
period+2
```

```
Period('2017-03', 'M')
```

To create a time series, you will need to create a sequence of dates. To create a sequence of Timestamps, use the pandas function **date_range**. You need to specify a start date, and/or an end date, or many periods. The default is daily frequency. The function returns the sequence of dates as a **DateTimeIndex** with frequency information. You will recognize the first element as a pandas Timestamp.

This is shown in the example below, and the output is shown in the figure below:

```
index = pd.date_range(start='2017-1-1', periods=12, freq='M')
index
 DatetimeIndex(['2017-01-31', '2017-02-28', '2017-03-31', '2017-04-30',
                '2017-05-31', '2017-06-30', '2017-07-31', '2017-08-31',
                '2017-09-30', '2017-10-31', '2017-11-30', '2017-12-31'],
               dtype='datetime64[ns]', freq='M')
```

1.2. Indexing & resampling time series

The basic transformations include parsing dates provided as strings and converting the result into the matching Pandas data type called datetime64. They also include selecting subperiods of your time series and setting or changing the frequency of the DateTimeIndex. You can change the frequency to a higher or lower value: upsampling involves increasing the time frequency, which requires generating new data. Downsampling means decreasing the time-frequency, which requires aggregating data.

To understand more about the transformations, we will apply this to the **Google stock prices data**. First, if you check the type of the date column, it is an object, so we would like to convert it into a date type by the following code.

```
google = pd.read_csv('google.csv') # import pandas as pd
google.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1094 entries, 0 to 1093
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Date    1094 non-null   object
 1   Close   756 non-null    float64
dtypes: float64(1), object(1)
memory usage: 17.2+ KB
```

The date information is converted from a string (object) into a datetime64, and we will also set the Date column as an index for the data frame, as it makes it easier to deal with the data by using the following code:

```
google.Date = pd.to_datetime(google.Date)
google.set_index('date', inplace=True)
google.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1094 entries, 0 to 1093
Data columns (total 2 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Date    1094 non-null   datetime64[ns]
 1   Close   756 non-null    float64
dtypes: datetime64[ns](1), float64(1)
memory usage: 17.2 KB
```

To have a better intuition of what the data looks like, let's plot the prices with time in Figure 1.1 using the code below:

```
google.Close.plot(title='Google Stock Price')
plt.tight_layout(); plt.show()
```

You can also perform partial indexing of the data using the date index, as in the following example:

```
google['2015'].info() # Pass string for part of date
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 365 entries, 2015-01-01 to 2015-12-31
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Close   252 non-null    float64
dtypes: float64(1)
memory usage: 5.7 KB
```

Figure 1.1. The change in Google's stock price over time.

You may have noticed that our DateTimeIndex did not have frequency information. You can set the frequency information using **dot-asfreq**. The alias 'D' stands for calendar day frequency. As a result, the DateTimeIndex now contains many dates where the stock wasn't bought or sold. The example below shows converting the DateTimeIndex of the Google stock data into calendar day frequency:

```
google.asfreq('D').info() # set calendar day frequency
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 1094 entries, 2014-01-02 to 2016-12-30
Freq: D
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Close   756 non-null    float64
```

The number of instances has increased to 756 due to this daily sampling. The code below prints the first five rows of the daily resampled data:

```
google.asfreq('D').head()
```

Date	Close
2014-01-02	556.00
2014-01-03	551.95
2014-01-04	NaN
2014-01-05	NaN
2014-01-06	558.10

We can see that some NaN values are missing new data due to this daily resampling.
We can also set the DateTimeIndex to business day frequency using the same method, but changing '**D**' into '**B**' in the **.asfreq()** method. This is shown in the example below:

```
google = google.asfreq('B') # Change to calendar day frequency
google.info()
```

If we print the first five rows, it will be as shown in the figure below:

Date	Close
2014-01-02	556.00
2014-01-03	551.95
2014-01-06	558.10
2014-01-07	568.86
2014-01-08	570.04

Now, the data available is only the working day's data.

1.3. Lags, changes, and returns for stock price series

Shift or lag values back or forward in time. shift(): Moving data between past & future. The default is one period into the future, but you can change it by giving the periods variable the desired shift value. An example of the shift method is shown below:

```
google['shifted'] = google.Close.shift() # default: periods=1
```

```
google.head(3)
```

The output is shown in the figure below:

Date	Close	shifted
2014-01-02	556.00	NaN
2014-01-03	551.95	556.00
2014-01-06	558.10	551.95

To move the data into the past, you can use **periods = -1** as shown below:

```
google['lagged'] = google.Close.shift(periods=-1)
google[['Close', 'lagged', 'shifted']].tail(3)
```

Date	Close	lagged	shifted
2016-12-28	785.05	782.79	791.55
2016-12-29	782.79	771.82	785.05
2016-12-30	771.82	NaN	782.79

One of the important properties of the stock prices data, and in general in the time series data, is the percentage change. Its formula is: $(\frac{x(t)}{x(t-1)} - 1) \times 100$. There are two ways to calculate it: we can use the built-in function **df.pct_change()** or use the functions **df.div.sub().mul(),** and both will give the same results as shown in the example below:

```
google['change'] = google.Close.div(google.shifted)
google['return'] = google.change.sub(1).mul(100)
google[['Close', 'shifted', 'change','return']].head(3)
```

Date	Close	shifted	change	return
2014-01-02	556.00	NaN	NaN	NaN
2014-01-03	551.95	556.00	0.992716	-0.728417
2014-01-06	558.10	551.95	1.011142	1.114231

```
google['pct_change'] = google.Close.pct_change().mul(100)
google[['Close', 'return', 'pct_change']].head(3)
```

Date	Close	return	pct_change
2014-01-02	556.00	NaN	NaN
2014-01-03	551.95	-0.728417	-0.728417
2014-01-06	558.10	1.114231	1.114231

We can also get multiperiod returns using the **periods** variable in the **df.pct_change()** method as shown in the following example.

```
google['return_3d'] = google.Close.pct_change(periods=3).mul(100)
google[['Close', 'return_3d']].head()
```

Date	Close	return_3d
2014-01-02	556.00	NaN
2014-01-03	551.95	NaN
2014-01-06	558.10	NaN
2014-01-07	568.86	2.312950
2014-01-08	570.04	3.277471

2. Basic Time Series Metrics & Resampling

In this section, we will dive deeper into the essential time-series functionality made available through the pandas DataTimeIndex. We will introduce resampling and how to compare different time series by normalizing their start points.

2.1. Resampling

We will start with resampling, which is changing the frequency of the time series data. This is a very common operation because you often need to convert two time series to a common frequency to analyze them together.

When you upsample by converting the data to a higher frequency, you create new rows and need to tell pandas how to fill or interpolate the missing values in these rows. When you downsample, you reduce the number of rows and need to tell pandas how to aggregate existing data.

To illustrate what happens when you up-sample your data, let's create a Series at a relatively low quarterly frequency for the year 2016 with the integer values 1–4. When you choose a quarterly frequency, pandas default to December for the end of the fourth quarter, which you could modify by using a different month with the quarter alias.

```
# Creating Quarterly data
dates = pd.date_range(start='2016', periods=4, freq='Q')
data = range(1, 5)
quarterly = pd.Series(data=data, index=dates)
quarterly
```

Next, let's see what happens when you up-sample your time series by converting the frequency from quarterly to monthly using **dot-asfreq()**. Pandas adds new month-end dates to the DateTimeIndex between the existing dates. As a result, there are now several months with missing data between March and December.

```
monthly = quarterly.asfreq('M') # to month-end frequency
monthly = monthly.to_frame('baseline') # to DataFrame
monthly
```

Let's compare three ways that pandas offer to fill missing values when upsampling. The first two options involve choosing a fill method, either forward fill or backfill. The third option is to provide full value.

	baseline
2016-03-31	1.0
2016-04-30	NaN
2016-05-31	NaN
2016-06-30	2.0
2016-07-31	NaN
2016-08-31	NaN
2016-09-30	3.0
2016-10-31	NaN
2016-11-30	NaN
2016-12-31	4.0

```
# Different Upsampling fill methods
monthly['ffill'] = quarterly.asfreq('M', method='ffill') # bfill back
fill
monthly['bfill'] = quarterly.asfreq('M', method='bfill') # ffill :
forward fill
monthly['value'] = quarterly.asfreq('M', fill_value=0)
```

If you compare the results, you see that forward fill propagates any value into the future if the future contains missing values. Backfill does the same for the past, and fill_value just substitutes missing values.

If you want a monthly DateTimeIndex that covers the full year, you can use **dot-reindex**. Pandas align existing data with the new monthly values and produce missing values elsewhere. You can use the same fill options for **dot-reindex** as you just did for **dot-asfreq**.

	baseline	ffill	bfill	value
2016-03-31	1.0	1	1	1
2016-04-30	NaN	1	2	0
2016-05-31	NaN	1	2	0
2016-06-30	2.0	2	2	2
2016-07-31	NaN	2	3	0
2016-08-31	NaN	2	3	0
2016-09-30	3.0	3	3	3
2016-10-31	NaN	3	4	0
2016-11-30	NaN	3	4	0
2016-12-31	4.0	4	4	4

```
# using reindex
dates = pd.date_range(start='2016', periods=12, freq='M')
quarterly.reindex(dates)
```

```
2016-01-31    NaN
2016-02-29    NaN
2016-03-31    1.0
2016-04-30    NaN
2016-05-31    NaN
2016-06-30    2.0
2016-07-31    NaN
2016-08-31    NaN
2016-09-30    3.0
2016-10-31    NaN
2016-11-30    NaN
2016-12-31    4.0
Freq: M, dtype: float64
```

2.2. Upsampling & interpolation

The resample method follows a logic similar to **dot-groupby**: It groups data within a resampling period and applies a method to this group. It takes the value that results from this method and assigns a new date within the resampling period. The new date is determined by a so-called offset, and for instance, can be at the beginning or end of the period or a custom location.

You will use resample to apply methods that either fill or interpolate missing dates when up-sampling, or that aggregate when down-sampling. We will apply the resample method to the **monthly unemployment rate**. First, we will upload it and parse it using the **DATE** column and make it an index.

```
unrate = pd.read_csv('unrate.csv', parse_dates=['DATE'],
index_col='DATE')
unrate.head()
```

DATE	UNRATE
2010-01-01	9.8
2010-02-01	9.8
2010-03-01	9.9
2010-04-01	9.9
2010-05-01	9.6

The 85 data points imported using read_csv since 2010 have no frequency information. An inspection of the first rows shows that the data are reported for the first of each calendar month. So let's resample it by the start of each calendar month using both **dot-resample** and **dot-asfreq** methods.

```
unrate.asfreq('MS').info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 209 entries, 2000-01-01 to 2017-05-01
Freq: MS
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   UNRATE  209 non-null    float64
dtypes: float64(1)
```

```
unrate.resample('MS') # creates Resampler object
```

```
unrate.asfreq('MS').equals(unrate.resample('MS').asfreq())
```

```
True
```

Both of the methods are the same. When looking at resampling by month, we have so far focused on month-end frequency. In other words, after resampling, new data will be assigned the last calendar day for each month. There are, however, quite a few alternatives as shown in the table below:

Frequency	Alias	Sample Date
Calendar Month End	M	2017-04-30
Calendar Month Start	MS	2017-04-01
Business Month End	BM	2017-04-28
Business Month Start	BMS	2017-04-03

Table 1. Resampling Period & Frequency Offsets

Depending on your context, you can resample to the beginning or end of either the calendar or business month. Resampling implements the following logic: When up-sampling, there will be more resampling periods than data points. Each resampling period will have a given date offset, for instance, a month-end frequency.

You then need to decide how to create data for the new resampling periods. The new data points will be assigned to the date offsets. In contrast, when down-sampling, there are more data points than resampling periods. Hence, you need to decide how to aggregate your data to obtain a single value for each date offset.

Let's now use a quarterly series, real GDP growth. You see that there is again no frequency info, but the first few rows confirm that the data are reported for the first day of each quarter.

```
# Upload the GDP growth
gdp = pd.read_csv('gdp_growth.csv', parse_dates=['date'],
index_col='date')
gdp.info()
```

date	gdp_growth
2007-01-01	0.2
2007-04-01	3.1
2007-07-01	2.7
2007-10-01	1.4
2008-01-01	-2.7

We can use **dot-resample** to convert this series to month month-start frequency, and then use forward fill logic to fill the gaps. We're using **dot-add_suffix** to distinguish the column label from the variation that we'll produce next.

```
gdp_1 = gdp.resample('MS').ffill().add_suffix('_ffill')
gdp_1.head()
```

date	gdp_growth_ffill
2007-01-01	0.2
2007-02-01	0.2
2007-03-01	0.2
2007-04-01	3.1
2007-05-01	3.1

Resample also lets you interpolate the missing values, that is, fill in the values that lie on a straight line between existing quarterly growth rates. A look at the first few rows shows how to interpolate the average's existing values.

```
gdp_2 = gdp.resample('MS').interpolate().add_suffix('_inter')
gdp_2.head()
```

date	gdp_growth_inter
2007-01-01	0.200000
2007-02-01	1.166667
2007-03-01	2.133333
2007-04-01	3.100000
2007-05-01	2.966667

We'll now combine the two series using the pandas **dot-concat** function to concatenate the two data frames. Using axis=1 makes pandas concatenate the DataFrames horizontally, aligning the row index. A plot of the data for the last two years visualizes how the new data points lie on the line between the existing points, whereas forward filling creates a step-like pattern.

```
# Plot interpolated real GDP growth
pd.concat([gdp_1, gdp_2], axis=1).loc['2015':].plot()
```

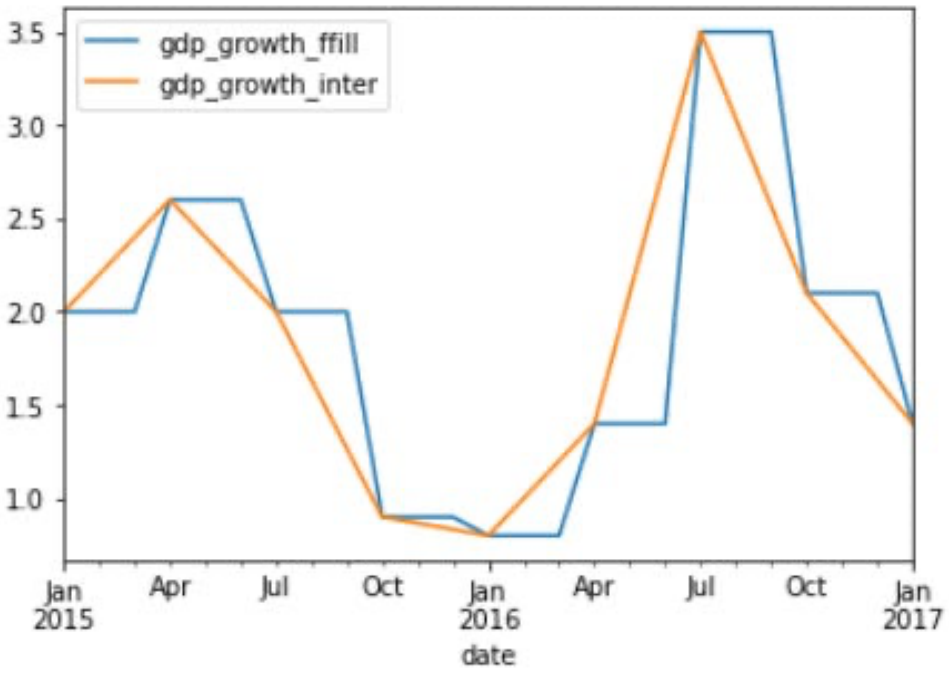


Figure 2.. The resampled GDP growth

After resampling GDP growth, you can plot the unemployment and GDP series based on their common frequency.

```
# Combine GDP growth & unemployment
pd.concat([unrate, gdp_2], axis=1).loc['2007':].plot();
```

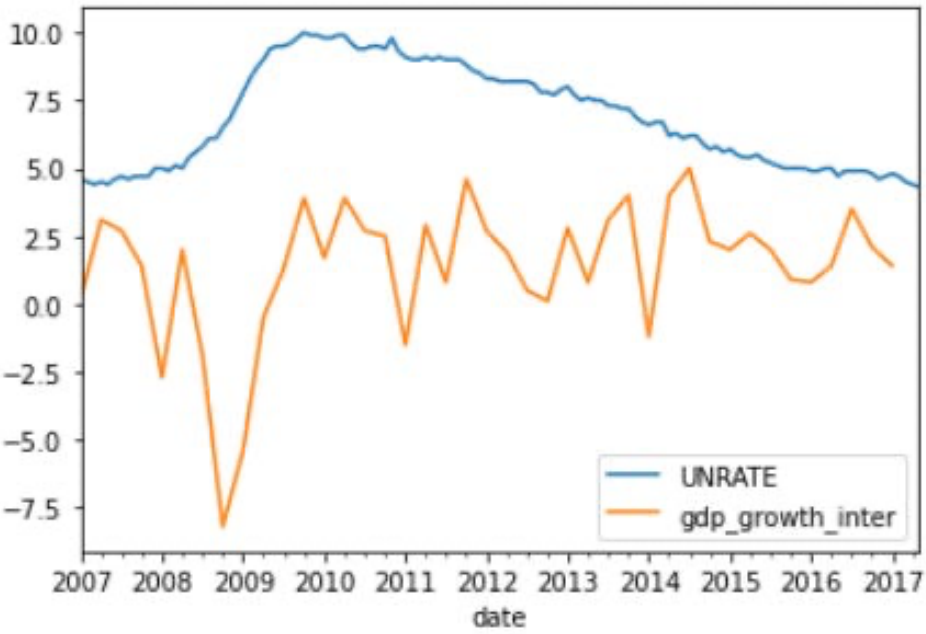


Figure 3. The unemployment and GDP series are based on their common frequency.

2.3. Downsampling & aggregation

So far, we have focused on up-sampling, that is, increasing the frequency of a time series, and how to fill or interpolate any missing values. Downsampling is the opposite is how to reduce the frequency of the time series data.

This includes, for instance, converting hourly data to daily data or daily data to monthly data. In this case, you need to decide how to summarize the existing data as 24 hours becomes a single day. Your options are familiar aggregation metrics like the mean or median, or simply the last value, and your choice will depend on the context.

Let's first use **read_csv** to import **air quality data** from the Environmental Protection Agency. It contains the average daily ozone concentration for New York City starting in 2000. Since the imported DateTimeIndex has no frequency, let's first assign calendar day frequency using **dot-resample**. The resulting DateTimeIndex has additional entries, as well as the expected frequency information.

```
ozone = pd.read_csv('ozone_nyc.csv', parse_dates=['date'],
index_col='date')
ozone.info()
```

```
DatetimeIndex: 6291 entries, 2000-01-01 to 2017-03-31
Data columns (total 1 columns):
 #   Column  Non-Null Count  Dtype
---  ------  --------------  -----
 0   Ozone   6167 non-null   float64
dtypes: float64(1)
memory usage: 98.3 KB
```

```
ozone = ozone.resample('D').asfreq()
ozone.info()
```

To convert daily ozone data to a monthly frequency, just apply the resample method with the new sampling period and offset. We are choosing a monthly frequency with a default month-end offset. Next, apply the mean method to aggregate the daily data to a single monthly value. You can see that the monthly average has been assigned to the last day of the calendar month.

```
ozone.resample('M').mean().head()
```

	Ozone
date	
2000-01-31	0.010443
2000-02-29	0.011817
2000-03-31	0.016810
2000-04-30	0.019413
2000-05-31	0.026535

You can apply the median in the same fashion.

```
ozone.resample('M').median().head()
```

	Ozone
date	
2000-01-31	0.009486
2000-02-29	0.010726
2000-03-31	0.017004
2000-04-30	0.019866
2000-05-31	0.026018

Similar to the **groupby** method, you can also apply multiple aggregations at once.

```
ozone.resample('M').agg(['mean', 'std']).head()
```

	Ozone	
	mean	std
date		
2000-01-31	0.010443	0.004755
2000-02-29	0.011817	0.004072
2000-03-31	0.016810	0.004977
2000-04-30	0.019413	0.006574
2000-05-31	0.026535	0.008409

Let's visualize the resampled, aggregated Series relative to the original data at calendar-daily frequency. We'll plot the data starting from 2016 so you can see more detail. Matplotlib allows you to plot several times on the same object by referencing the axes object that contains the plot.

```
ozone = ozone.loc['2016':]
ax = ozone.plot()
monthly = ozone.resample('M').mean()
monthly.add_suffix('_monthly').plot(ax=ax)
```

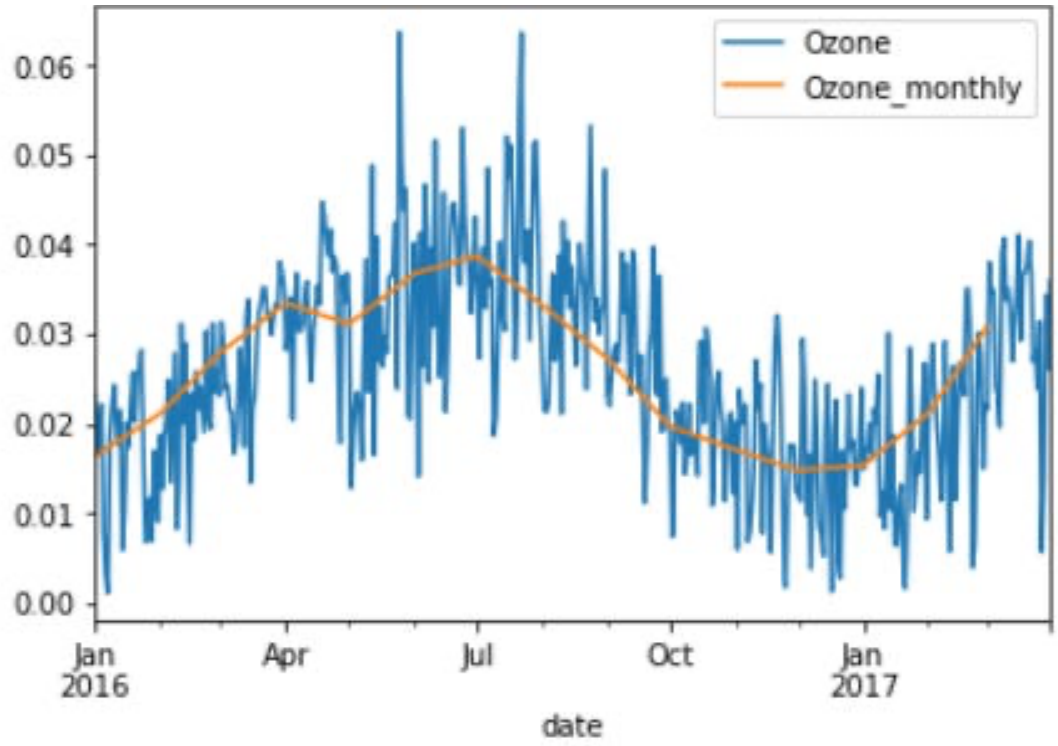


Figure 4. The resampled, aggregated Series relative to the original data.

The blue line represents the original series, and the orange line represents the resampled series with a suffix so that the legend reflects the difference. You see that the resampled data are much smoother since the monthly volatility has been averaged out. Let's also take a look at how to resample several series.

3. Window Functions: Rolling & Expanding Metrics

In this section, we will show you how to use the window function to calculate time series metrics for both rolling and expanding windows.

3.1. Window functions with pandas

Window functions are useful because they allow you to operate on sub-periods of your time series. In particular, window functions calculate metrics for the data inside the window. Then, the result of this calculation forms a new time series, where each data point represents a summary of several data points of the original time series.

We will discuss two main types of windows: **Rolling windows** maintain the same size while they slide over the time series, so each new data point is the result of a given number of observations. **Expanding windows** grow with the time series so that the calculation that produces a new data point is the result of all previous data points.

Let's calculate a simple moving average to see how this works in practice. We will again use Google stock price data for the last several years. We will see two ways to define the rolling window.

First, we apply **rolling** with an integer window size of 30. This means that the window will contain the previous 30 observations or trading days. When you choose an integer-based window size, pandas will only calculate the mean if the window has no missing values. You can change this default by setting the min_periods parameter to a value smaller than the window size of 30.

```
# Integer-based window size
data.rolling(window=30, min_periods=1).mean() # fixed # observations
```

	Close
Date	
2014-01-02	556.000000
2014-01-03	553.975000
2014-01-04	553.975000
2014-01-05	553.975000
2014-01-06	555.350000
...	...
2016-12-26	779.575500
2016-12-27	780.145714
2016-12-28	780.946190
2016-12-29	781.515238
2016-12-30	782.171429

1094 rows × 1 columns

You can also create windows based on a date offset. If you choose 30D, for instance, the window will contain the days when stocks were traded during the last 30 calendar days. While the

window is fixed in terms of period length, the number of observations will vary. Let's take a look at what the rolling mean looks like.

```
# Offset-based window size
data.rolling(window='30D').mean() # fixed period length
```

Date	Close
2014-01-02	556.000000
2014-01-03	553.975000
2014-01-04	553.975000
2014-01-05	553.975000
2014-01-06	555.350000
...	...
2016-12-26	779.575500
2016-12-27	780.145714
2016-12-28	780.946190
2016-12-29	781.515238
2016-12-30	782.171429

1094 rows × 1 columns

You can also calculate a 90 calendar day rolling mean, and join it to the stock price. The join method allows you to concatenate a Series or DataFrame along axis 1, that is, horizontally. It's just a different way of using the **dot-concat** function you've seen before. You can see how the new time series is much smoother because every data point is now the average of the preceding 90 calendar days.

```
r90 = data.rolling(window='90D').mean()
data.join(r90.add_suffix('_mean_90')).plot()
```

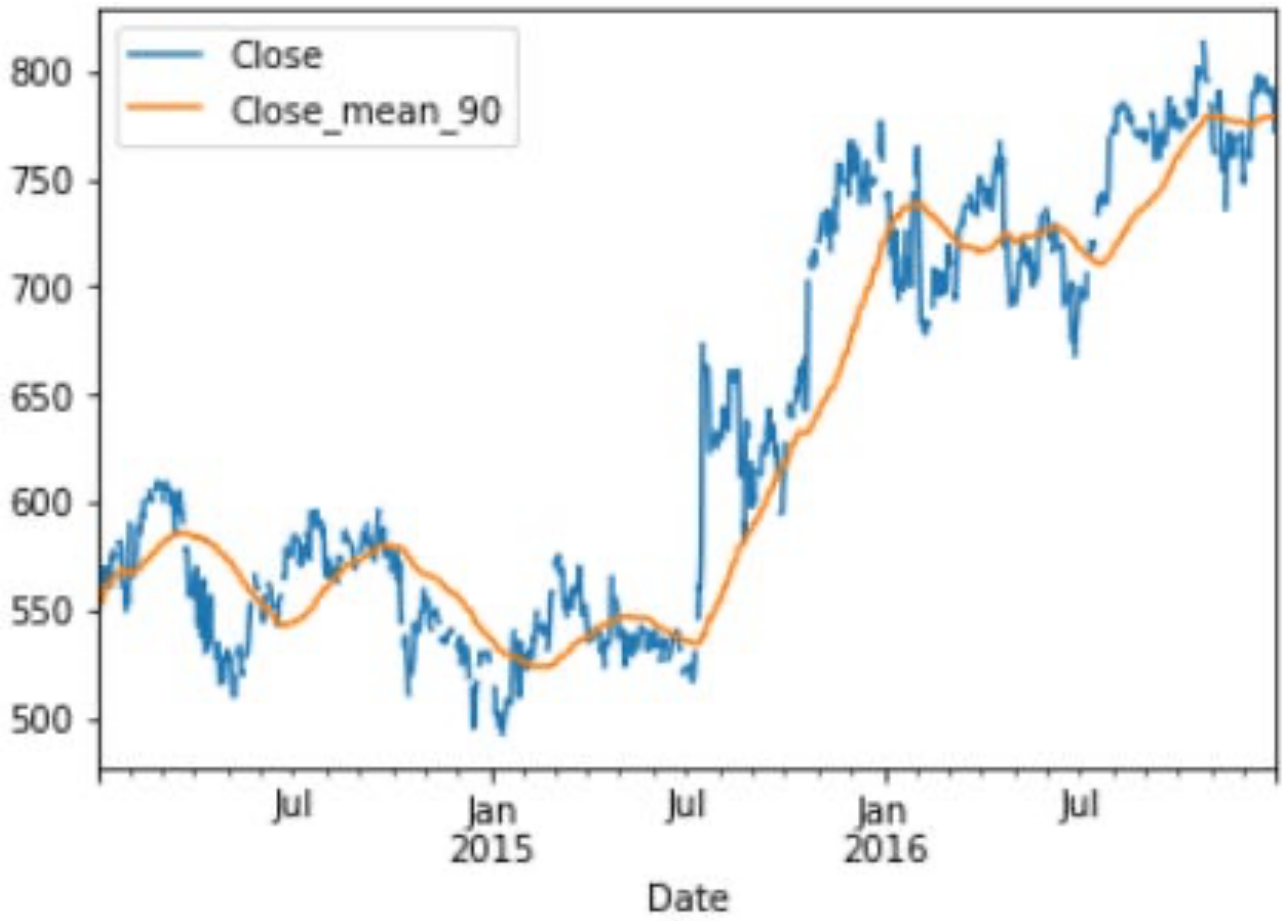


Figure 1.5. The price vs the mean price over 90 days.

To see how extending the time horizon affects the moving average, let's add the 360 calendar moving average nad plot it in Figure 1.6.

```
data['mean90'] = r90
r360 = data['Close'].rolling(window='360D').mean()
data['mean360'] = r360
data.plot()
```

The series now appears smoother still, and you can more clearly see when short-term trends deviate from longer-term trends, for instance, when the 90-day average dips below the 360-day average in 2015.

Similar to **dot-groupby**, you can also calculate multiple metrics at the same time, using the **dot-agg** method. With a 90-day moving average and standard deviation, you can easily discern periods of heightened volatility. This is shown in Figure 1.7.

```
r = data.Close.rolling('90D').agg(['mean', 'std'])
r.plot(subplots = True)
```

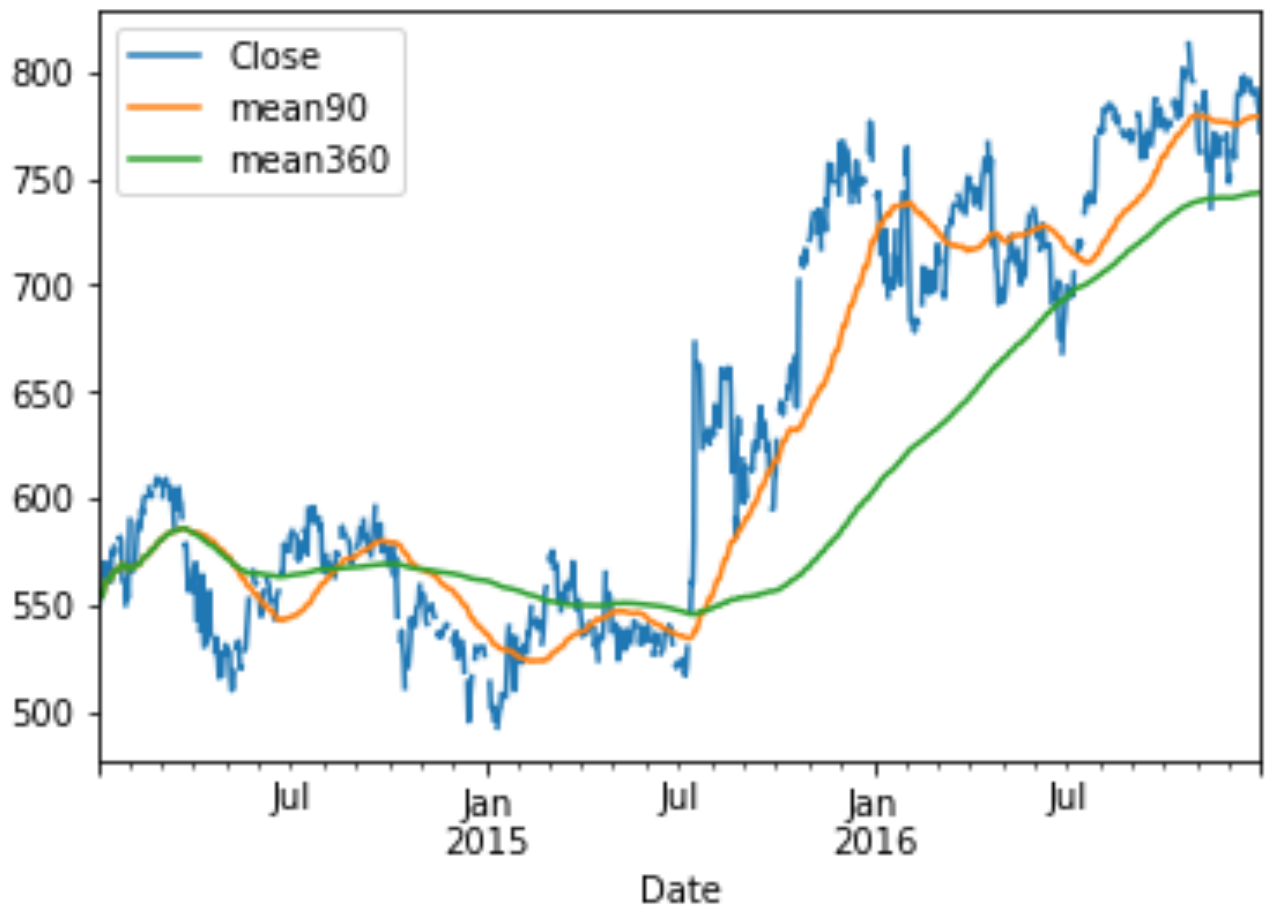


Figure 1.6. The 90 and 360 mean rolling average.

Finally, let's display a 360-day rolling median, or 50th percentile, alongside the 10th and 90th percentiles in Figure 1.8. Again, you can see how the ranges for the stock price have evolved, with some periods more volatile than others.

```
rolling = data.Close.rolling('360D')
q10 = rolling.quantile(0.1).to_frame('q10')
median = rolling.median().to_frame('median')
q90 = rolling.quantile(0.9).to_frame('q90')
pd.concat([q10, median, q90], axis=1).plot()
```

3.2. Expanding window functions with pandas

We will move from rolling to expanding windows. You will now calculate metrics for groups that get larger to exclude all data up to the current date. Each data point of the resulting time series reflects all historical values up to that point. Expanding windows are useful to calculate, for instance, a cumulative rate of return, or a running maximum or minimum. In pandas, you can

use either the expanding method, which works just like rolling, or, in a few cases, the shorthand methods for the cumulative sum, product, min, and max.

We will use the S&P 500 **data** for the last ten years in the practical examples in this section. Let's first take a look at how to calculate returns: The simple period return is just the current price divided by the last price minus 1. The return over several periods is the product of all period returns after adding 1 and then subtracting 1 from the product.

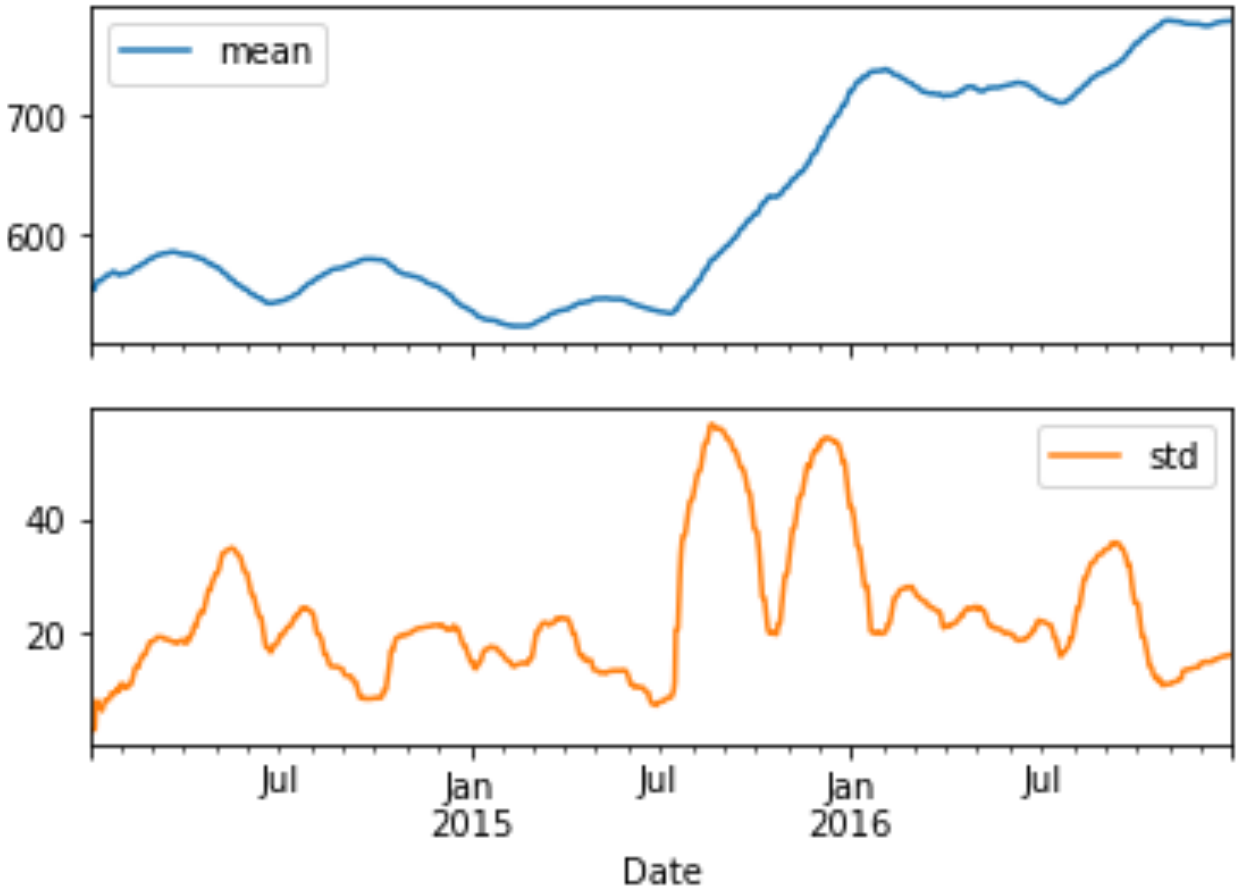


Figure 1.7. The 90-day moving average and standard deviation.

So for more clarification, the period return is: ***r(t) = (p(t)/p(t-1)) -1*** and the multi-period return is: ***R(T) = (1+r(1))(1+r(2))……..(1+r(T))—1***. Pandas makes these calculations easy, as you have already seen the methods for percent change(.pct_change) and basic math (.diff(), .div(), .mul()), and now you'll learn about the cumulative product.

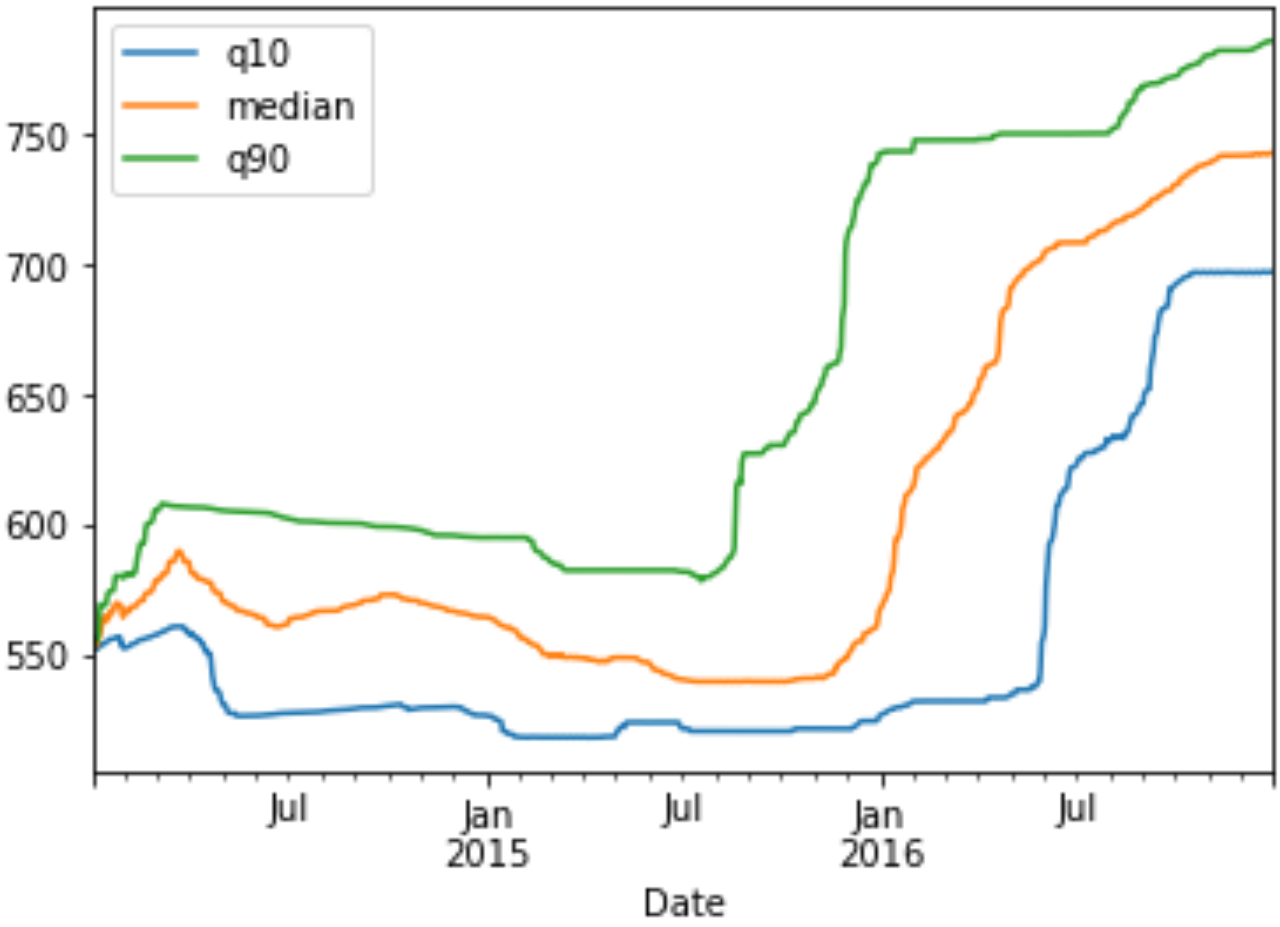


Figure 1.8. The 90-calendar day rolling median, 10 and 90 percent quantiles.

To calculate the cumulative or running rate of return on the S&P 500, follow the steps described above: calculate the period return with a percent change and add 1. Calculate the cumulative product, and subtract one. You can multiply the result by 100 and plot the result in percentage terms. The code for this is shown below:

```
pr = data.SP500.pct_change() # period return
pr_plus_one = pr.add(1)
cumulative_return = pr_plus_one.cumprod().sub(1)
cumulative_return.mul(100).plot()
```

From Figure 1.9, we can see that the SP500 is up 60% since 2007, despite being down 60% in 2009. You can also easily calculate the running min and max of a time series: Just apply the expanding method and the respective aggregation method.

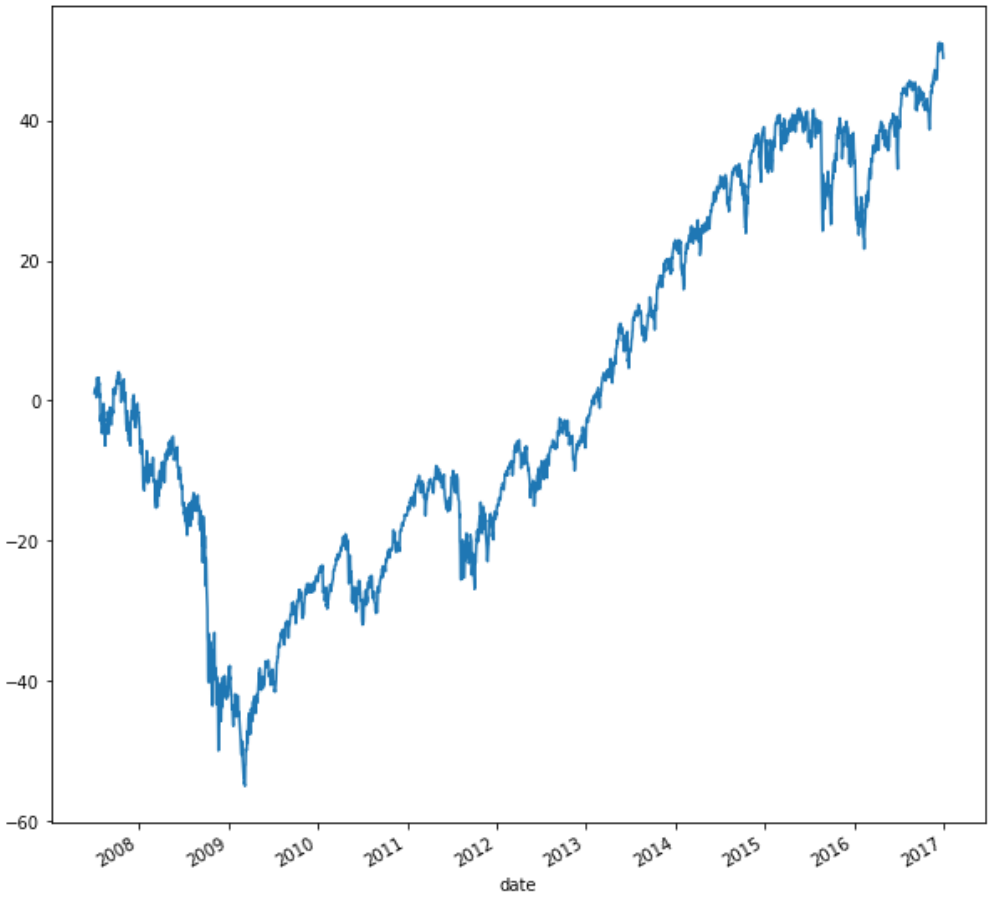


Figure 1.9. The cumulative or running rate of return on the S&P 500

```
data['running_min'] = data.SP500.expanding().min()
data['running_max'] = data.SP500.expanding().max()
data.plot(figsize=(10,10)
```

The orange and green lines outline the min and max up to the current date for each day. You can also combine the concept of a rolling window with a cumulative calculation. Let's calculate the rolling annual rate of return, that is, the cumulative return for all 360 calendar day periods over the ten-year period covered by the data.

This cumulative calculation is not available as a built-in method. But no problem, just define your own multiperiod function, and **apply** it to run it on the data in the rolling window. The data in the rolling window is available to your multi_period_return function as a **numpy** array. Add 1 to increment all returns, apply the **numpy** product function, and subtract one to implement the formula from above.

Figure 1.10. The running min and max of the SP 500 time series

Just pass this function to apply after creating a 360 calendar day window for the daily returns. Multiply the rolling 1-year return by 100 to show them in percentage terms, and plot alongside the index using subplots equals True.

```
def multi_period_return(period_returns):
    return np.prod(period_returns + 1) - 1

pr = data.SP500.pct_change() # period return
r = pr.rolling('360D').apply(multi_period_return)
data['Rolling 1yr Return'] = r.mul(100)
data.plot(subplots=True, figsize=(10,10))
```

Figure 1.11. The SP500, running min and max, and the rolling 1-year return.

The results show large annual return swings following the 2008 crisis.

3.4. Random walk and simulations

Daily stock returns are notoriously hard to predict, and models often assume they follow a random walk. We will use NumPy to generate random numbers in a time series context. You'll also use the cumulative product again to create a series of prices from a series of returns.

In the first example, we will generate random numbers from the bell-shaped normal distribution. This means that values around the average are more likely than extremes, as tends to be the case with stock returns. In the second example, you will randomly select actual S&P 500 returns to then simulate S&P 500 prices.

To generate random numbers, first import the normal distribution and the seed functions from the numpy module random. Also, import the norm package from scipy to compare the normal distribution alongside your random samples.

Generate 1000 random returns from numpy's normal function, and divide by 100 to scale the values appropriately. Let's plot the distribution of the 1,000 random returns and fit a normal distribution to your sample. You can see that the sample closely matches the shape of the normal distribution.

```
# Generate random numbers

import seaborn as sns
from numpy.random import normal, seed
from scipy.stats import norm
seed(42)
random_returns = normal(loc=0, scale=0.01, size=1000)
sns.distplot(random_returns, fit=norm, kde=False)
```

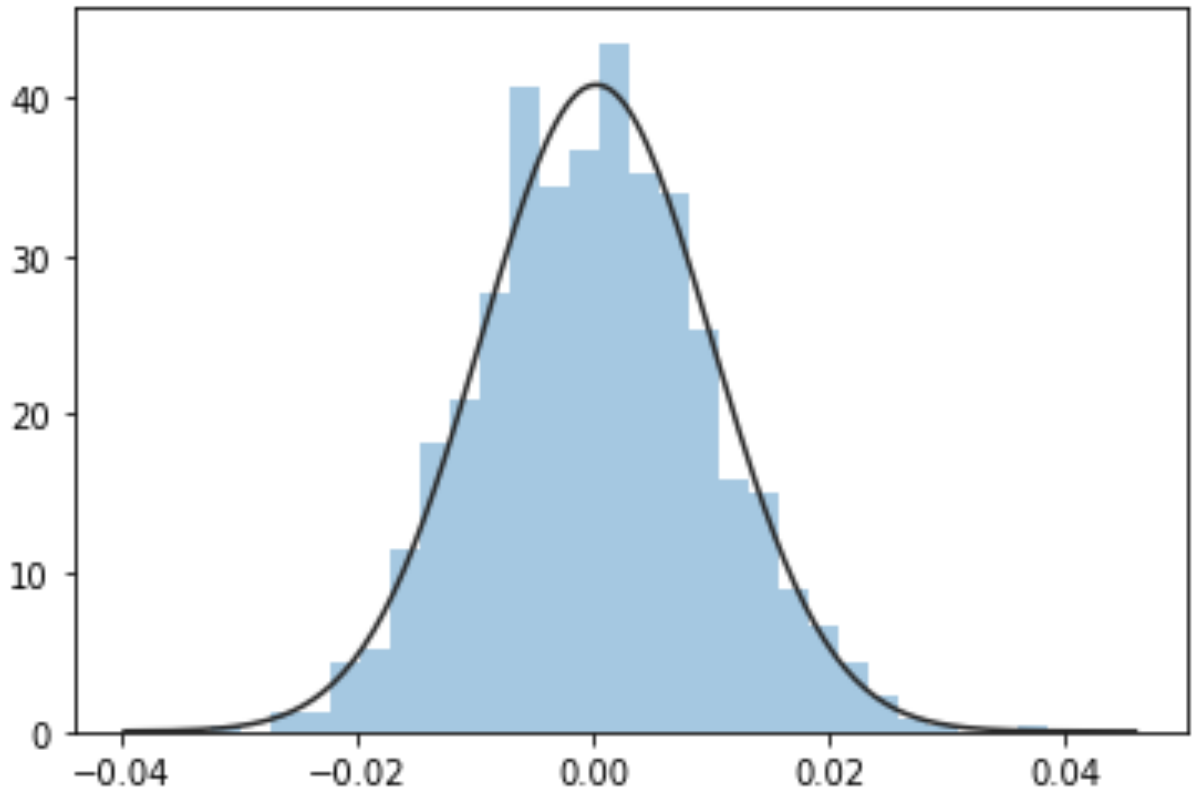


Figure 1.12. Normal distribution and 1000 random returns

To create a random price path from your random returns, we will follow the procedure from the subsection, after converting the numpy array to a pandas Series. Add 1 to the period returns, calculate the cumulative product, and subtract 1. Plot the cumulative returns, multiplied by 100, and you see the resulting prices.

```
# Create a random price path
return_series = pd.Series(random_returns)
random_prices = return_series.add(1).cumprod().sub(1)
random_prices.mul(100).plot(figsize=(10,5))
```

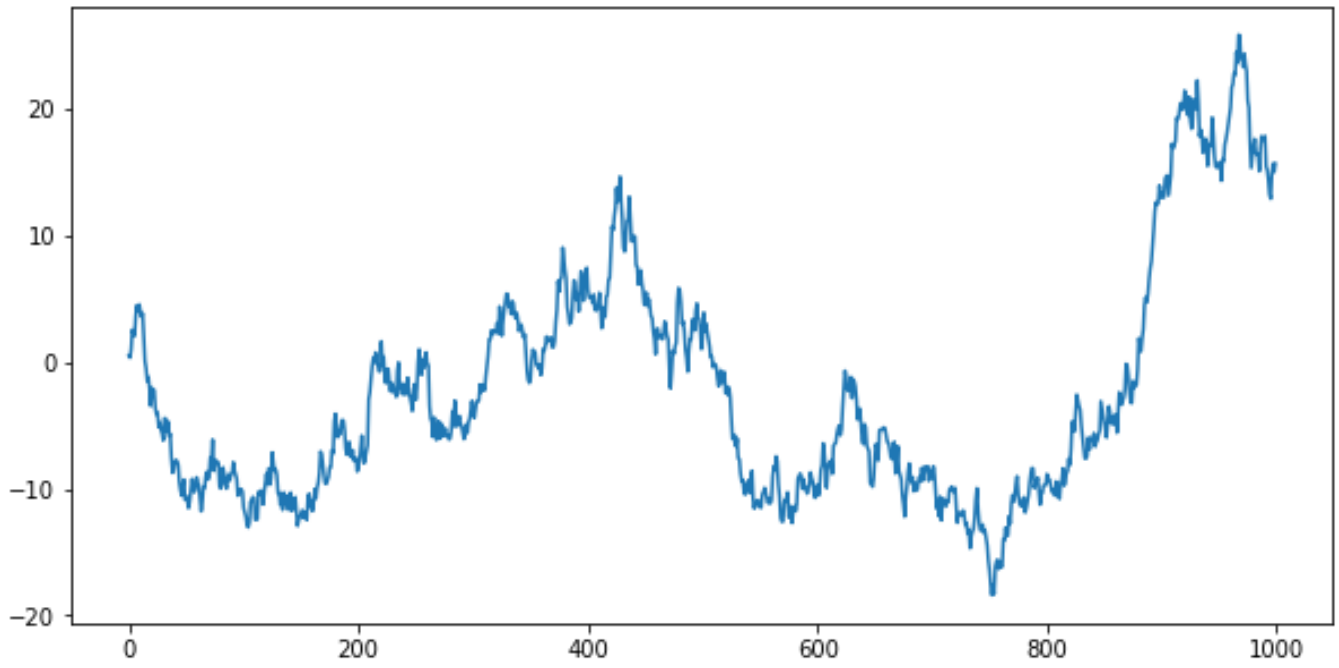


Figure 1.13. The random returns.

Let's now simulate the SP500 using a random expanding walk. Import the last 10 years of the index, drop missing values, and add the daily returns as a new column to the DataFrame. A plot of the index and return series shows the typical daily return range between +/2–3 percent, as well as a few outliers during the 2008 crisis.

```
# S&P 500 prices & returns
data = pd.read_csv('sp500.csv', parse_dates=['date'],
index_col='date')
data['returns'] = data.SP500.pct_change()
data.plot(subplots=True, figsize=(10,10))
```

A comparison of the S&P 500 return distribution to the normal distribution shows that the shapes don't match very well. This is a typical finding: daily stock returns tend to have outliers more often than the normal distribution would suggest.

Now let's randomly select from the actual S&P 500 returns. You'll be using the choice function from Numpy's random module. It returns a NumPy array with a random sample from a list of numbers, in our case, the S&P 500 returns. Just provide the return sample and the number of observations you want to the choice function.

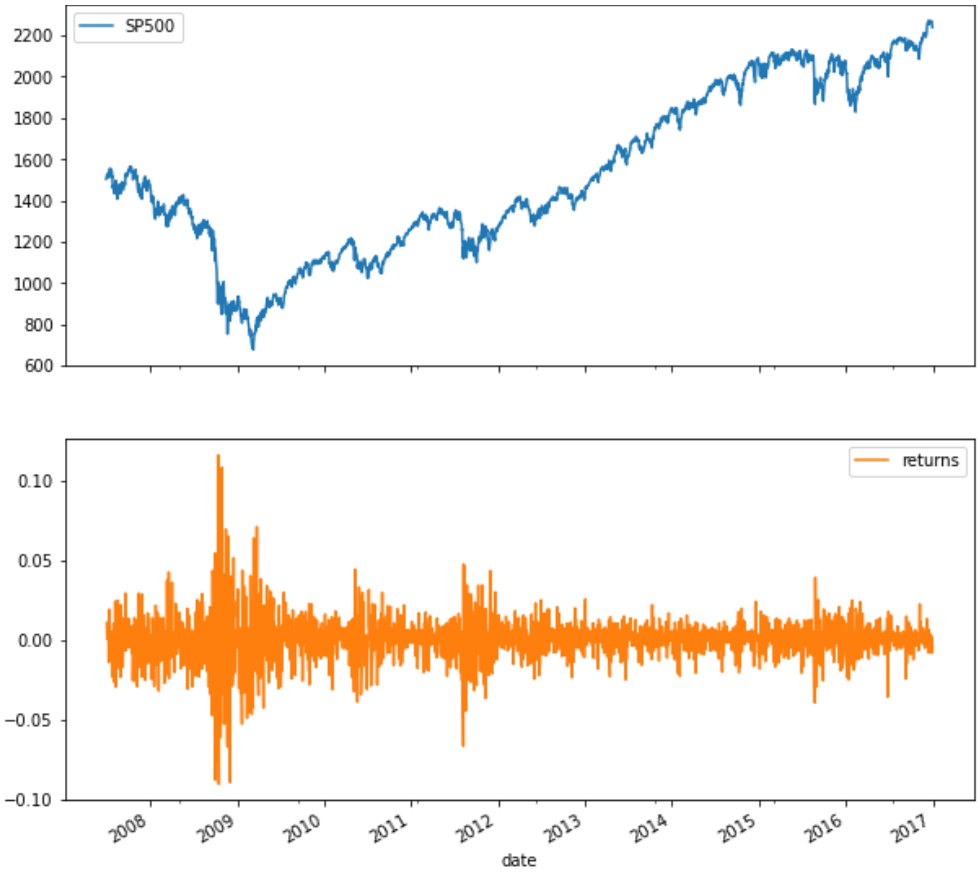


Figure 1.14. The S&P 500 prices & returns

Next, convert the NumPy array to a pandas series, and set the index to the dates of the S&P 500 returns. Your random walk will start at the first S&P 500 price.

```
from numpy.random import choice
sample = data.returns.dropna()
n_obs = data.returns.count()
random_walk = choice(sample, size=n_obs)
random_walk = pd.Series(random_walk, index=sample.index)
random_walk.head()
```

Use the '**first**' method with a calendar day offset to select the first S&P 500 price. Then add 1 to the random returns, and append the return series to the start value. Now you are ready to calculate the cumulative return given the actual S&P 500 start value. Add 1, calculate the cumulative product, and subtract one. The result is a random walk for the SP500 based on random samples from actual returns.

```
# Random S&P 500 prices
start = data.SP500.first('D')
sp500_random = start.append(random_walk.add(1))
data['SP500_random'] = sp500_random.cumprod()
data[['SP500', 'SP500_random']].plot(figsize=(10,10))
```

Figure 1.15. The random walk for the SP500 is based on random samples from actual returns.

3.4. Correlation between time series

Correlation is the key measure of linear relationships between two variables. In financial markets, correlations between asset returns are important for predictive models and risk management, for instance. Pandas and seaborn have various tools to help you compute and visualize these relationships.

The correlation coefficient looks at pairwise relations between variables and measures the similarity of the pairwise movements of two variables around their respective means. This pairwise co-movement is called covariance.

The correlation coefficient divides this measure by the product of the standard deviations for each variable. As a result, the coefficient varies between -1 and +1. The closer the correlation coefficient is to plus 1 or minus 1, the more a plot of the pairs of the two series resembles a straight line.

The sign of the coefficient implies a positive or negative relationship. A positive relationship means that when one variable is above its mean, the other is likely also above its mean, and vice versa for a negative relationship. **There are, however, numerous types of non-linear relationships that the correlation coefficient does not capture.**

We will use this ***price series*** for five assets to analyze their relationships in this section. You now have 10 years' worth of data for two stock indices, a bond index, oil, and gold.
Seaborn has a joint plot that makes it very easy to display the distribution of each variable together with the scatter plot that shows the joint distribution.

We'll use the daily returns for our analysis. The joint plot takes a DataFrame and then two column labels for each axis. The S&P 500 and the bond index, for example, have low correlation given the more diffuse point cloud and negative correlation as suggested by the slight downward trend of the data points.

```
daily_returns = data.pct_change()
sns.jointplot(x='SP500', y='Bonds', data=daily_returns)
```

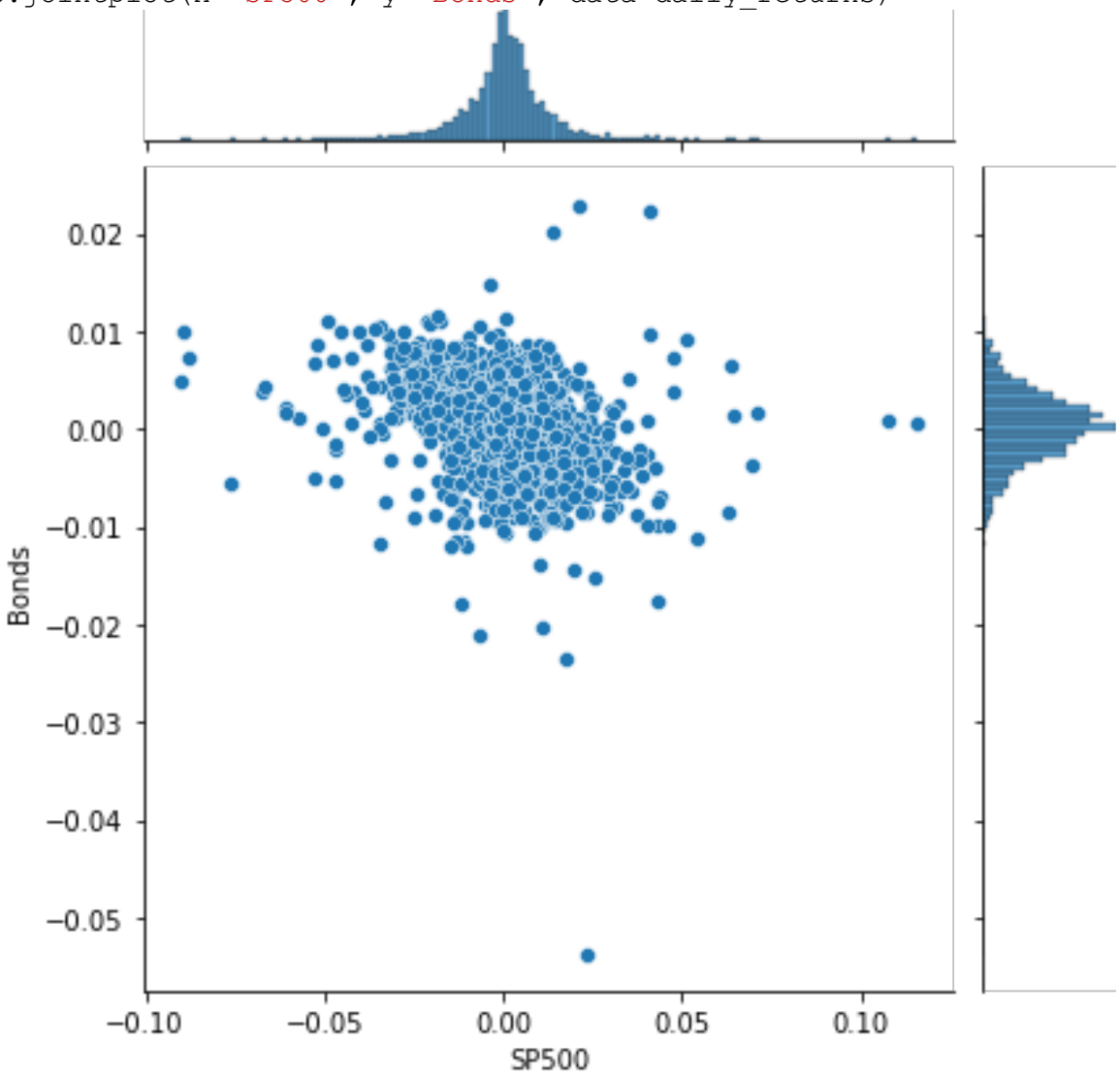


Figure 1.16. The correlation coefficient between the bond index

Pandas allow you to calculate all pairwise correlation coefficients with a single method called **dot-corr**. Apply it to the returns DataFrame, and you get a new DataFrame with the

pairwise coefficients. The data are naturally symmetric around the diagonal, which contains only values of 1 because the correlation of a variable with itself is, of course, 1.

```
correlations = daily_returns.corr()
correlations
```

	SP500	Bonds	Gold	Oil
SP500	1.000000	-0.314890	-0.006546	0.335353
Bonds	-0.314890	1.000000	0.028882	-0.215279
Gold	-0.006546	0.028882	1.000000	0.104272
Oil	0.335353	-0.215279	0.104272	1.000000

Seaborn again offers a neat tool to visualize pairwise correlation coefficients. The heatmap takes the DataFrame with the correlation coefficients as inputs and visualizes each value on a color scale that reflects the range of relevant values. The parameter annot equals True ensures that the values of the correlation coefficients are displayed as well. You can see that the correlations of daily returns among the various asset classes vary quite a bit.

Figure 17. Pairwise correlation coefficients heatmap between Oil, Bonds, Gold, and SP500.

4. Putting it all together: Building a value-weighted index

This chapter combines the previous concepts by teaching you how to create a value-weighted index. This index uses market-cap data contained in the stock exchange listings to calculate weights and 2016 stock price information.

Index performance is then compared against benchmarks to evaluate the performance of the index you created. To build a value-based index, you will take several steps: You will select the largest company from each sector using actual stock exchange data as index components.

Then, you'll calculate the number of shares for each company and select the matching stock price series from a file. Next, you'll compute the weights for each company, and based on these, the index for each period. You will also evaluate and compare the index performance.

4.1 Select index components & import data

First, let's import company data using pandas' read_excel function. You will import **this worksheet** with listing info from a particular exchange while making sure missing values are properly recognized.

```
# Load stock listing data
nyse = pd.read_excel('listings.xlsx', 2, na_values='n/a')
nyse.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3147 entries, 0 to 3146
Data columns (total 7 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   Stock Symbol          3147 non-null   object
 1   Company Name          3147 non-null   object
 2   Last Sale             3079 non-null   float64
 3   Market Capitalization 3147 non-null   float64
 4   IPO Year              1361 non-null   float64
 5   Sector                2177 non-null   object
 6   Industry              2177 non-null   object
dtypes: float64(3), object(4)
memory usage: 172.2+ KB
```

Next, move the stock ticker into the index. Since you'll select the largest company from each sector, remove companies without sector information. You can use the 'subset' keyword to identify one or several columns to filter out missing values.

You have already seen the keyword 'inplace' to avoid creating a copy of the DataFrame. Finally, divide the market capitalization by 1 million to express the values in million USD. The results are 2177 companies from the NYSE stock exchange.

```
# prepare stock listing data
nyse.set_index('Stock Symbol', inplace=True)
nyse.dropna(subset=['Sector'], inplace=True)
nyse['Market Capitalization'] /= 1e6 # in Million USD
```

To pick the largest company in each sector, group these companies by sector, select the column market capitalization, and apply the method **nlargest** with parameter 1. The result is a Series with the market cap in millions with a MultiIndex.

```
# Select index components
components = nyse.groupby(['Sector'])['Market
Capitalization'].nlargest(1)
components.sort_values(ascending=False)
```

The first index level contains the sector, and the second is the stock ticker. To select the tickers from the second index level, select the series index, and apply the method '**get_level_values**' with the name of the index '**Stock Symbol**'. You can also use the value 1 to select the second index level. Print the tickers, and you see that the result is a single DataFrame index. Use the method **dot-tolist** to obtain the result as a list.

```
tickers = components.index.get_level_values('Stock Symbol')
tickers.tolist()
```

Finally, use the ticker list to select your stocks from a broader set of recent price time series imported using read_csv.

```
tickers = components.index.get_level_values('Stock Symbol')
tickers.tolist()
```

4.2 Build a market-cap weighted index

To construct the **market-cap** weighted index, you need to calculate the number of shares using both market capitalization and the latest stock price, because the market capitalization is just the product of the number of shares and the price of each share.

Next, you'll use the historical stock prices to convert them into a series of market values. Then convert it to an index by normalizing the series to start at 100. You'll also take a look at the index return and the contribution of each component to the result.

To calculate the number of shares, just divide the market capitalization by the last price. Since we are measuring market cap in million USD, you obtain the shares in millions as well. You can now multiply your historical stock price series by the number of shares.

```
shares = component_info['Market
Capitalization'].div(component_info['Last Sale'])
data = pd.read_csv('stocks.csv', parse_dates=['Date'],
index_col='Date').loc[:, tickers.tolist()]
market_cap_series = data.mul(len(shares))
market_cap_series.info()
```

The result is a time series of the market capitalization, ie, the stock market value of each company. By selecting the first and the last day from this series, you can compare how each company's market value has evolved over the year.

```
# From stock prices to market value
market_cap_series.first('D').append(market_cap_series.last('D'))
```

Now you almost have your index: just get the market value for all companies per period using the sum method with the parameter axis equals 1 to sum each row.

```
# Aggregate market value per period
agg_mcap = market_cap_series.sum(axis=1) # Total market cap
agg_mcap.plot(title='Aggregate Market Cap')
```

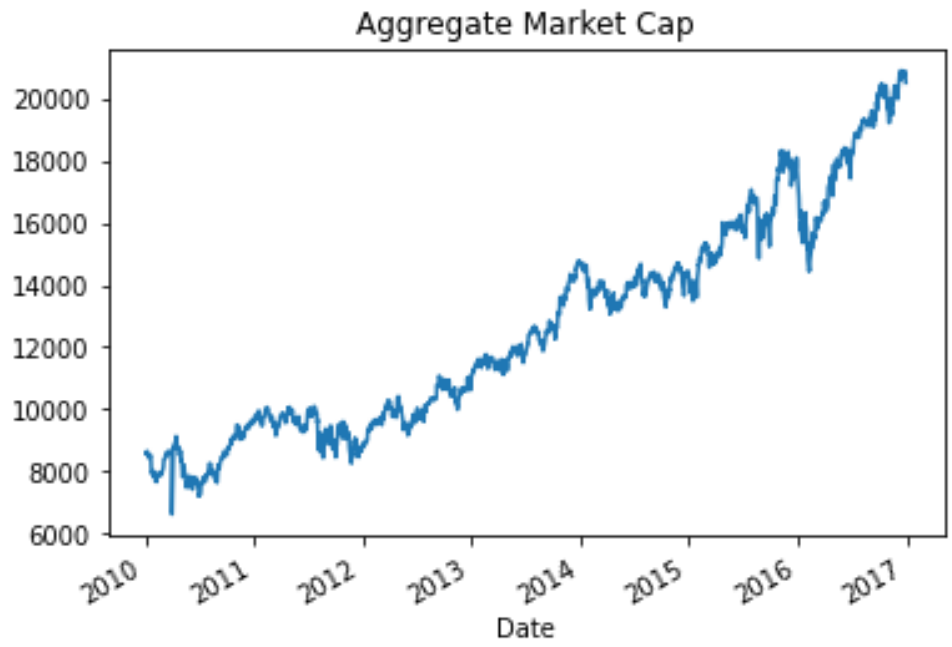


Figure 18. The aggregate market value per period.

Now you just need to normalize this series to start at 1 by dividing the series by its first value, which you get using **dot-iloc**. Multiply the result by 100, and you get the convenient start value of 100, where differences from the start values are changes in percentage terms.

```
# Value-based index
index = agg_mcap.div(agg_mcap.iloc[0]).mul(100) # Divide by 1st value
index.plot(title='Market-Cap Weighted Index')
```

4.3. Evaluate index performance

Now that you have built a weighted index, you can analyze its performance. Important elements of your analysis will be: First, take a look at the index return and the contribution of each component to the result.

Next, compare the performance of your index to a benchmark like the S&P 500, which covers the wider market and is also value-weighted. You can compare the overall performance or rolling returns for sub-periods.

First, let's look at the contribution of each stock to the total value-added over the year. Subtract the last value of the aggregate market cap from the first to see that the companies in the index added 315 billion dollars in market cap.

To see how much each company contributed to the total change, apply the diff method to the last and first value of the series of market capitalization per company and period.
The last row now contains the total change in market cap since the first day. You can select the last row using dot-loc and the date of the last row, or **iloc** with the parameter -1.

```
# Value contribution by stock
```

```
change =
market_cap_series.first('D').append(market_cap_series.last('D'))
change.diff().iloc[-1].sort_values() # or: .loc['2016-12-30']
```

To compute the contribution of each component to the index return, let's first calculate the component weights. Select the market capitalization for the index components. Calculate the component weights by dividing their market cap by the sum of the market caps of all components. As you can see, the weights vary between 2 and 13%. Now, calculate the total index return by dividing the last index value by the first value, subtracting 1, and multiplying by 100.

```
# Market-cap based weights
market_cap = component_info['Market Capitalization']
weights = market_cap.div(market_cap.sum())
weights.sort_values().mul(100)
```

Let's now move on and compare the composite index performance to the S&P 500 for the same period. Convert the index series to a DataFrame so you can insert a new column. Import the data from the Federal Reserve as before.

Then normalize the S&P 500 to start at 100 just like your index, and insert it as a new column, then plot both time series. You can see that your index did a couple of percentage points better for the period.

```
# Performance vs benchmark
data = index.to_frame('Index') # Convert pd.Series to pd.DataFrame

data['SP500'] = pd.read_csv('sp500.csv', parse_dates=['date'],
index_col='date')

data.SP500 = data.SP500.div(data.SP500.iloc[0], axis=0).mul(100)
data.plot(figsize=(10,8))
```

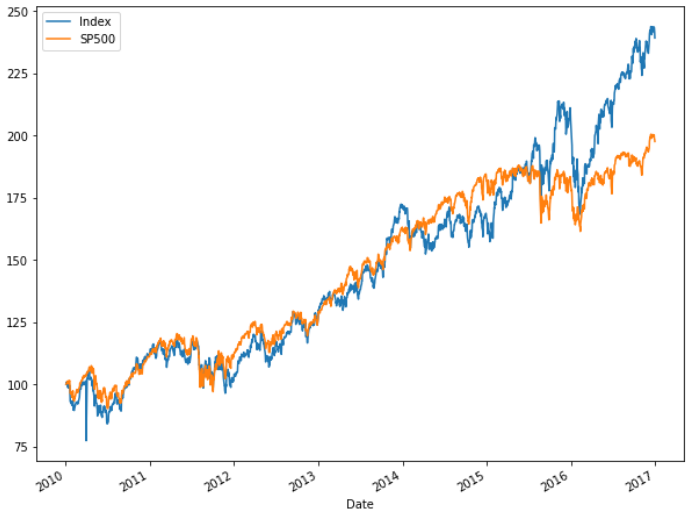


Figure 19. The composite index performance relative to the S&P 500.

Lastly, to compare the performance over various subperiods, create a multi-period-return function that compounds a **NumPy array** of period returns to a multi-period return as you did in section 3.

Create the daily returns of your index and the S&P 500, a 30 calendar day rolling window, and apply your new function. The plot shows all 30-day returns for either series and illustrates when it was better to be invested in your index or the S&P 500 for 30 days.

```
def multi_period_return(r):
    return (np.prod(r + 1) - 1) * 100
data.pct_change().rolling('30D').apply(multi_period_return).plot()
```

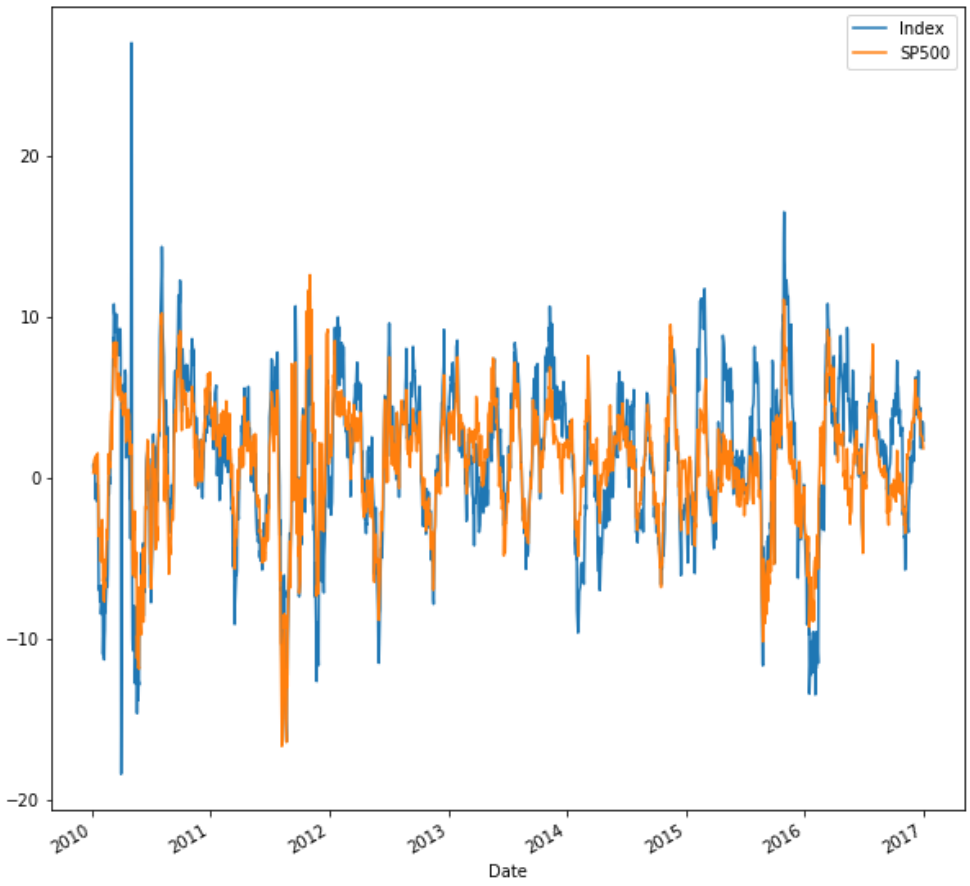


Figure 20. The 30-day returns for the S&P 500 and the index.

Time Series Data Analysis In Python

Table of Contents:

1. Correlation and Autocorrelation

In this section, you'll be introduced to the ideas of correlation and autocorrelation for time series. Correlation describes the relationship between two time series, and autocorrelation describes the relationship of a time series with its past values.

1.1. Correlation of Two Time Series

The correlation of the two time series measures how they vary with each other. The correlation coefficient summarizes this relation in one number. A correlation of one means that the two series have a perfect linear relationship with no deviations.

High correlations mean that the two series strongly vary together. A low correlation means they vary together, but there is a weak association. And a high negative correlation means they vary in opposite directions, but still with a linear relationship.

There is a common mistake when calculating the correlation between two trending time series. Consider two time series that are both trending. Even if the two series are totally unrelated, you could still get a very high correlation. That's why, when you look at the correlation of, say, two stocks, you should look at the correlation of their **returns**, not their **levels**.

In the example below, the two series, **stock prices and UFO sightings**, both trend up over time. Of course, there is no relationship between those two series, but the correlation is 0.94. But if you compute the correlation of percent changes, the correlation goes down to approximately zero.

```
# Compute correlation of levels
# data used is levels
levels = pd.read_csv('DJI.csv', parse_dates=['Date'],
index_col='Date')

correlation1 = levels['DJI'].corr(levels['UFO'])
print("Correlation of levels: ", correlation1)
```

```
# Compute correlation of percent changes
changes = levels.pct_change()
correlation2 = changes['DJI'].corr(changes['UFO'])
print("Correlation of changes: ", correlation2)
```

```
Correlation of levels:  0.9204594155244163
Correlation of changes:  0.009287288259357323
```

The figure below shows that the two series are correlated when plotted with time. The reason for this, as mentioned, is that they are both trending series.

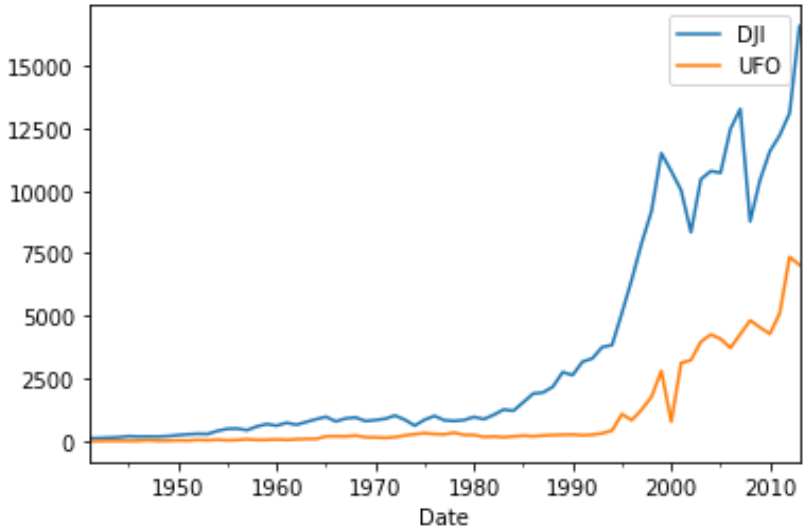


Figure 2.1. The change in the two series over time.

1.2. Simple Linear Regression

A simple linear regression for time series finds the slope, beta, and intercept, alpha, of a line that's the best fit between a dependent variable, y, and an independent variable, x. The x's and y's can be two time series.

Regression techniques are very common, and therefore, many packages in Python can be used. In statsmodels, there is OLS. In **numpy**, there is **polyfit**, and if you set degree equal to 1, it fits the data to a line, which is a linear regression.

Pandas has an **OLS** method, and **scipy** has a linear regression function. Beware that the order of x and y is not consistent across packages. In the example below, we will regress the values of the oil prices using the S&P 500 as an independent variable. The data can be found **here**. You need to add a column of ones as a dependent, right-hand side variable.

The reason you have to do this is that the regression function assumes that if there is no constant column, then you want to run the regression without an intercept. By adding a column of ones, **statsmodels** will compute the regression coefficient of that column as well, which can be interpreted as the intercept of the line. The statsmodels method "add constant" is a simple way to add a constant.

```
# Import the statsmodels module
import statsmodels.api as sm

# Compute correlation of x and y

data = pd.read_csv('asset.csv', parse_dates=['DATE'],
index_col='DATE')
data = data.dropna()
x = data['SP500']
y = data['Oil']

correlation = x.corr(y)
print("The correlation between x and y is %4.2f" %(correlation))

# Convert the Series x to a DataFrame and name the column x
dfx = pd.DataFrame(x.values, columns=['x'])

# Add a constant to the DataFrame dfx
dfx1 = sm.add_constant(dfx)

# Regress y on dfx1
result = sm.OLS(y.values, dfx1).fit()

# Print out the results and look at the relationship between R-squared
and the correlation above
print(result.summary())
```

The regression output is shown below.

```
The correlation between x and y is -0.35
                            OLS Regression Results
==============================================================================
Dep. Variable:                      y   R-squared:                       0.120
Model:                            OLS   Adj. R-squared:                  0.120
Method:                 Least Squares   F-statistic:                     336.5
Date:                Wed, 23 Mar 2022   Prob (F-statistic):          1.44e-70
Time:                        18:12:29   Log-Likelihood:                -11214.
No. Observations:                2469   AIC:                         2.243e+04
Df Residuals:                    2467   BIC:                         2.244e+04
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
const        108.1995      1.710     63.273      0.000     104.846     111.553
x             -0.0194      0.001    -18.345      0.000      -0.021      -0.017
==============================================================================
Omnibus:                       51.243   Durbin-Watson:                   0.007
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               29.897
Skew:                          -0.095   Prob(JB):                     3.22e-07
Kurtosis:                       2.496   Cond. No.                     6.05e+03
==============================================================================
```

We will only focus on the yellow-highlighted items of the regression results. The coef contains the slope and the intercept of the regression analysis. Since the two variables are negatively correlated, the slope is negative.

The second important statistic to take note of is the R-squared, which is 0.12. The R-squared measures how well the linear regression line fits the data. There is a relation between the correlation and the R-squared. The magnitude of the correlation is the square root of the R-squared. And the sign of the correlation is the sign of the slope of the regression line.

1.3. Autocorrelation

Autocorrelation is the correlation of a single time series with a lagged copy of itself. It's also called serial correlation. Often, when we refer to a series's autocorrelation, we mean the "lag-one" autocorrelation. So, when using daily data, for example, the autocorrelation would be the correlation of the series with the same series lagged by one day.

Positive autocorrelation, which is also known as trend following, means that the increase observed in a time interval leads to a proportionate increase in the lagged time interval.

Negative autocorrelation, which is known as mean-reverting, means that if a particular value is above average, the next value (or, for that matter, the previous value) is more likely to be below average. The figure below shows an example of both of them.

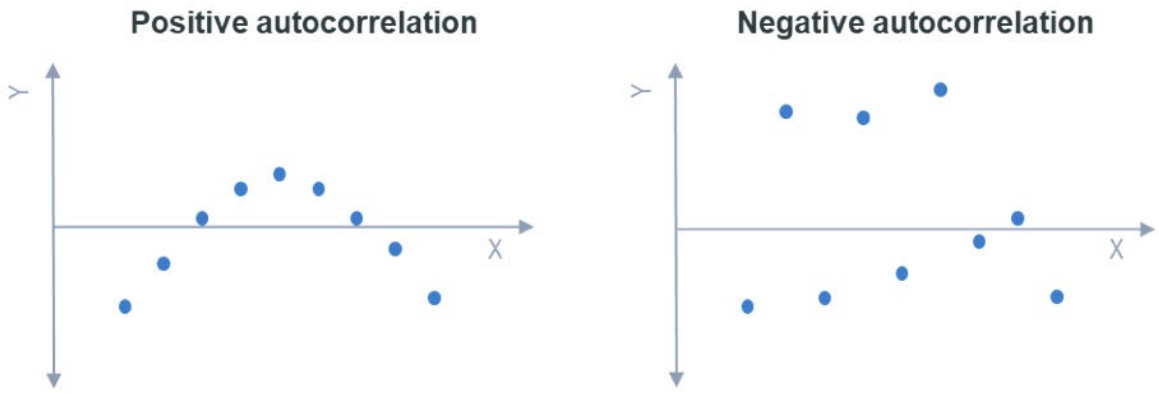


Figure 2.2. Positive Vs negative autocorrelation

The autocorrelation of time series has many real-world applications. Many hedge fund strategies are only slightly more complex versions of mean reversion and momentum strategies.
Since stocks have historically had negative autocorrelation over horizons of about a week, one popular strategy is to buy stocks that have dropped over the last week and sell stocks that have gone up.

For other assets like commodities and currencies, they have historically had positive autocorrelation over horizons of several months, so the typical hedge fund strategy there is to buy commodities that have gone up in the last several months and sell those commodities that have gone down.

1.4. Autocorrelation Function

The autocorrelation function (ACF) is the autocorrelation as a function of the lag. Any significant non-zero autocorrelations imply that the series can be forecast from the past.

It can be determined which values you should rely on to forecast the values in the future, discover seasonal earnings by observing the autocorrelation function at these seasons, and can be used for selecting a model for fitting the data, as will be shown in section 3.

plot_acf is the **statsmodels** function for plotting the autocorrelation function. The input x is a series or array. The argument lags indicates how many lags of the autocorrelation function will be plotted. The alpha argument sets the width of the confidence interval.

For example, if alpha equals 0.05, that means that if the true autocorrelation at that lag is zero, there is only a 5% chance the sample autocorrelation will fall outside that window. You will get a wider confidence interval if you set alpha lower or if you have fewer observations.

```
# Import the acf module and the plot_acf module from statsmodels
from statsmodels.tsa.stattools import acf
from statsmodels.graphics.tsaplots import plot_acf

# Compute the acf array of HRB
acf_array = acf(HRB)
print(acf_array)

# Plot the acf function
plot_acf(HRB, alpha=0.05)
plt.show()
```

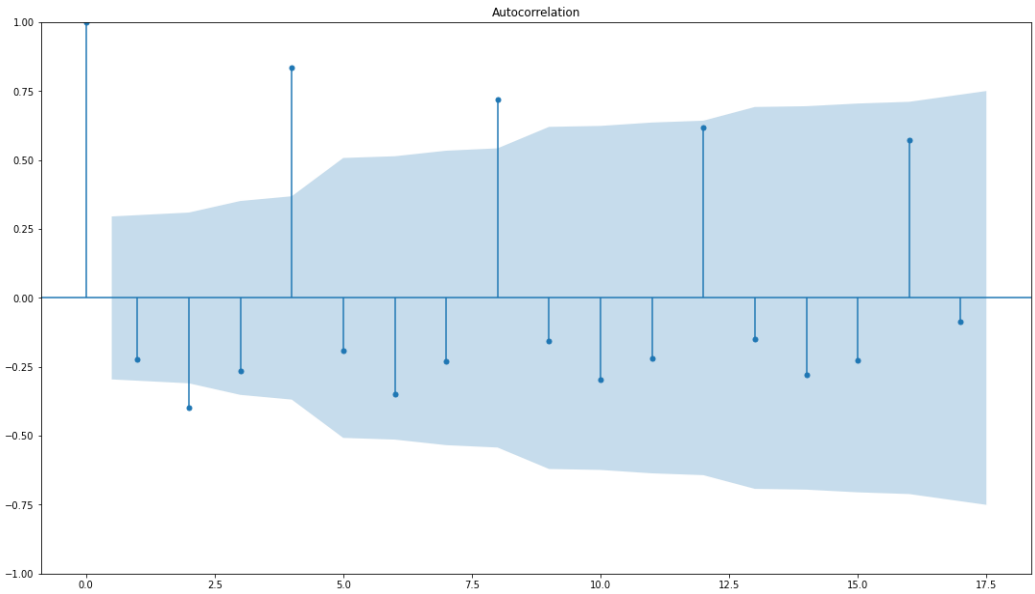


Figure 2.3. The auto-correlation function (ACF) plot.

If you want no bonds, set **alpha = 1**

2. Time Series Models

In this section, you'll learn about some simple time series models. These include white noise and a random walk.

2.1. White Noise

White noise is a series with constant mean and variance with time and zero autocorrelation at all lags. There are several special cases of white noise. For example, if the data is white noise but also has a normal, or Gaussian, distribution, then it is called Gaussian White Noise.

The white noise looks like the following:

```
noise = np.random.normal(loc=0, scale=1, size=500)
plt.plot(noise)
plt.show()
plot_acf(noise, lags=50)
plt.show()
```

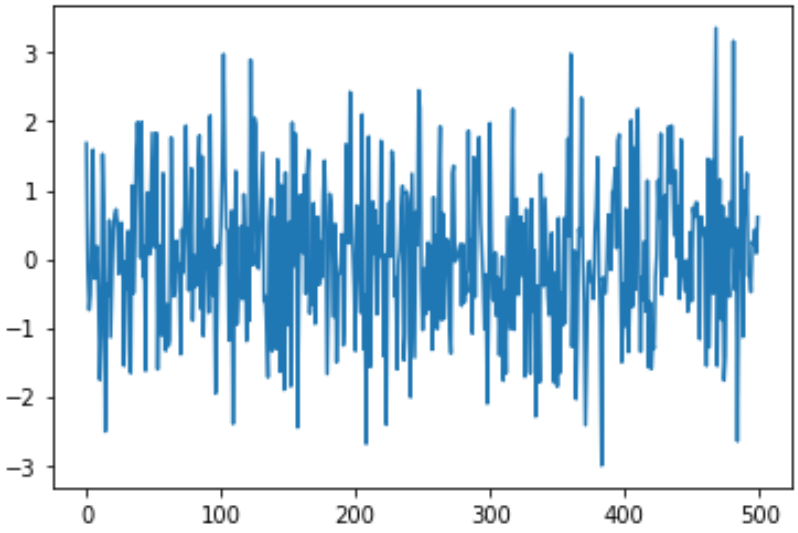


Figure 2.4. White noise example

The autocorrelation of the white noise:

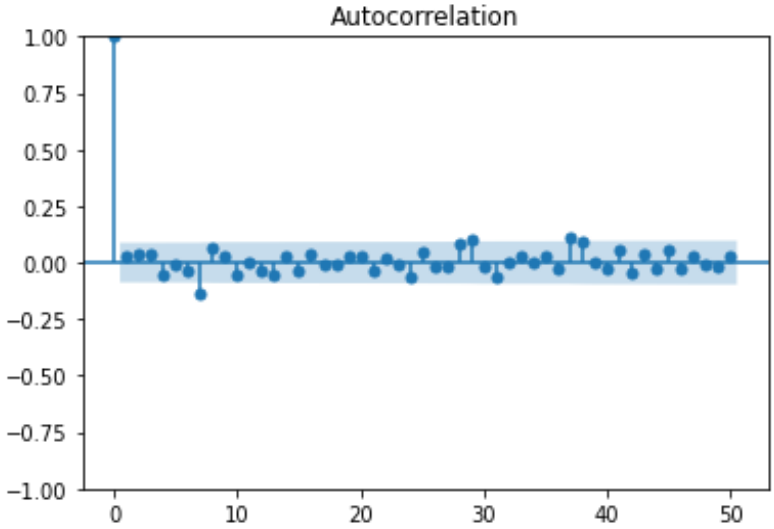


Figure 2.5. The autocorrelation of white noise.

2.2. Random Walk

In a random walk, today's price is the same as yesterday's price, in addition to white noise.

$$P(t) = P(t-1) + \text{white-noise}$$

Therefore, the change in the price is white noise. Since we cannot forecast the white noise, the best forecast for today's price will be yesterday's price.
To test whether a time series follows a random walk or not, we can regress the current value (for example, the price) on the lagged values. If the slope coefficient is significantly less than one, we can reject the null hypothesis (the series follows a random walk). If it is not significantly different from one, we cannot reject the null hypothesis.

Another way to do it is to regress the difference in values on the lag values and test the slope coefficient to be zero instead of one. This is known as **the Dickey-Fuller** test, and if more lagged values are added, it will be called the **Augmented Dickey-Fuller** (ADF) test.
The example below shows how to apply this test to the **SP500** data in Python using the **statsmodels** library.

```
from statsmodels.tsa.stattools import adfuller
data = pd.read_csv('asset.csv', parse_dates=['DATE'],
index_col='DATE')
data = data.dropna()
SPX = data['SP500']
results = adfuller(SPX)
print(results[1]) # print the p-value
```

The **p-value is 0.9**, which means that the difference is not significant and we cannot reject the null hypothesis; therefore, the SP500 time series follows a random walk.

2.3. Stationary

Stationery signals are the signals whose joint distribution does not depend on time. A more practical definition is the weak stationarity definition, which means that the mean, variance, and autocorrelation of the signal do not depend on time.

Stationery is a very important concept in time series analysis and forecasting. The reason for this is that if the series is not stationary, we would not be able to model and forecast it.
The main task of the modeling step is estimating a set of parameters that could be used for the required tasks, such as forecasting. If the series is non-stationary, its parameters will be changing over time, so you will not be able to model it.

A random walk is a common type of non-stationary series. The variance grows with time. For example, if stock prices are a random walk, then the uncertainty about prices tomorrow is much less than the uncertainty 10 years from now. Seasonal series are also non-stationary. Many non-stationary series can be made stationary through a simple transformation. A Random Walk is a non-stationary series, but if you take the first differences, the new series is White Noise, which is stationary.

In the example below, the **S&P 500 prices**, which are a non-stationary random walk, the signal is transformed into a stationary white noise signal by taking the first difference.

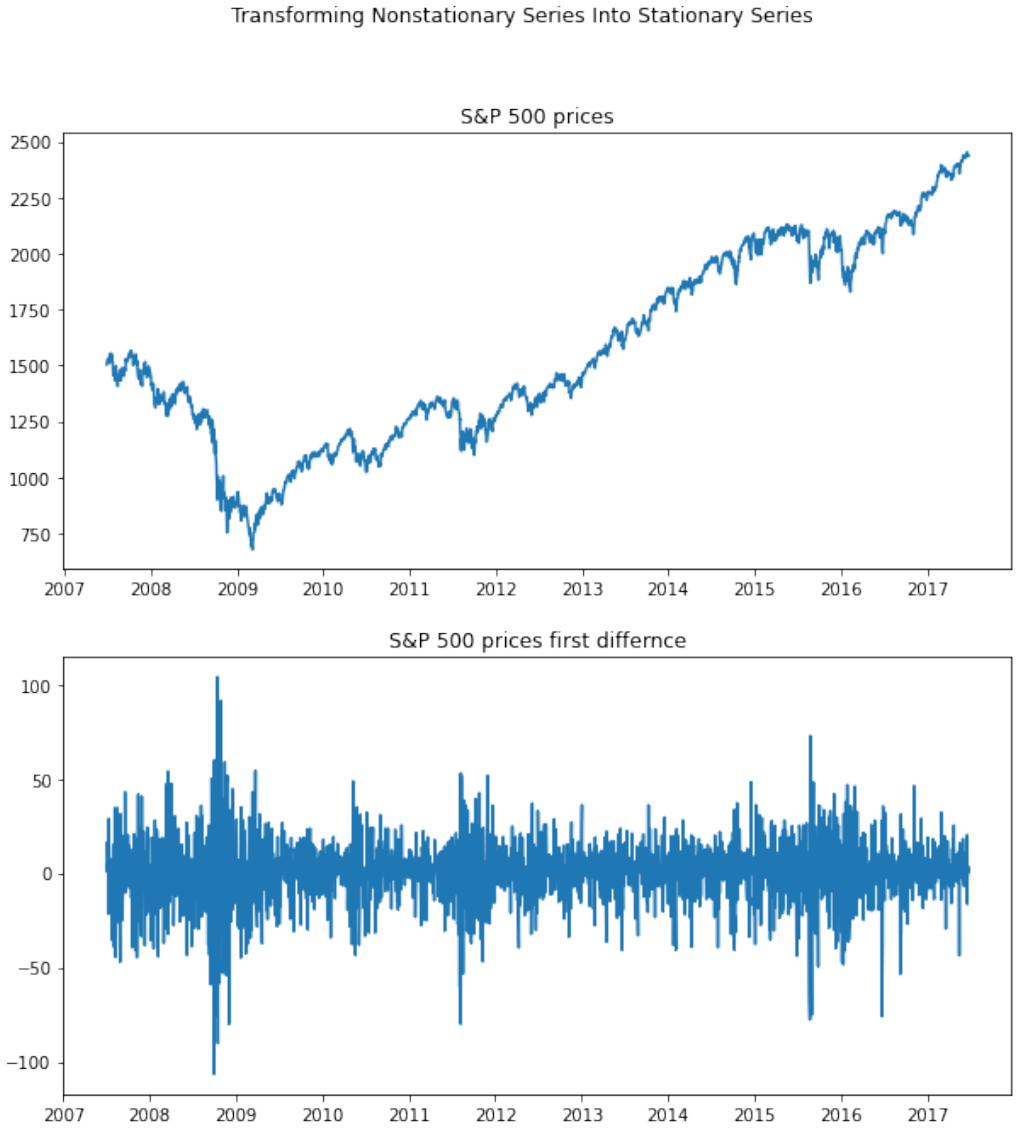


Figure 2.6. Transforming non-stationary series into stationary series.

The example below is for the quarterly earnings for **H&R Block**, which has a large seasonal component and is therefore not stationary. If we take the seasonal difference, by taking the difference with a lag of 4, the transformed series looks stationary.

```
# Transforming Nonstationary Series Into Stationary Series
fig, axs = plt.subplots(2)
fig.suptitle('Transforming Nonstationary Series Into Stationary
Series')
fig.set_size_inches(10, 10.5)
```

```
axs[0].plot(HRB)
axs[0].set_title('Quarterly earnings for H&R Block')

axs[1].plot(HRB.diff(4))
axs[1].set_title('Quarterly earnings for H&R Block seasonal
difference')
```

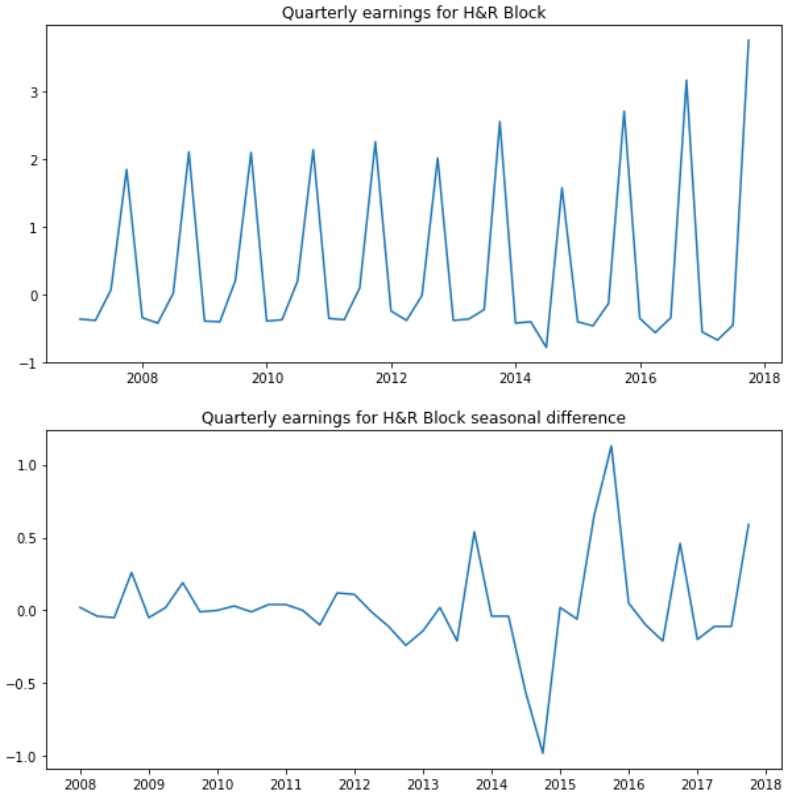


Figure 2.7. Transforming the quarterly earnings for H&R from non-stationary into stationary.

The last example is Amazon's quarterly revenue. It is growing exponentially as well as exhibiting a strong seasonal pattern. First, we will take only the log of the series to eliminate the exponential growth. But if you take both the log of the series and then the seasonal difference, the transformed series looks stationary.

```
# Transforming Nonstationary Series Into Stationary Series
AMZN =  pd.read_csv('AMZN.xls', parse_dates=['Date'],
index_col='Date')

fig, axs = plt.subplots(3)
fig.suptitle('Transforming Nonstationary Series Into Stationary
Series')
fig.set_size_inches(10, 10.5)

axs[0].plot(AMZN)
axs[0].set_title('Amazon quarterly revenue')
axs[1].plot(np.log(AMZN))
axs[1].set_title('Log of Amazon quarterly revenue')
```

```
axs[2].plot(np.log(AMZN).diff(4))
axs[2].set_title('Seasonal differnce of Amazon quarterly revenue log')
```

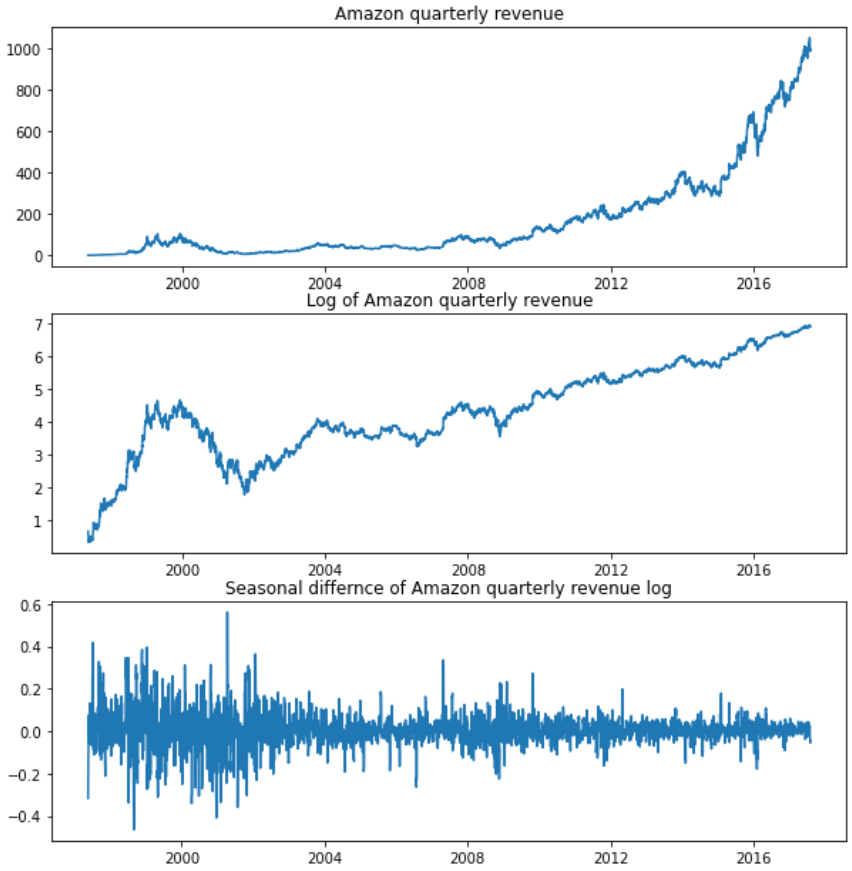


Figure 2.8. Amazon's quarterly revenue, the log, and the seasonal difference of Amazon quarterly revenue.

3. Autoregressive (AR) Models

In this section, we will explain the autoregressive, or AR, models for time series. These models use past values of the series to predict the current value.

3.1. AR Models Definition

An autoregressive (AR) model **predicts future behavior based on past behavior**. It's used for forecasting when there is some correlation between values in a time series and the values that precede and succeed them.

In the AR model, today's value is equal to the fraction (phi) of yesterday's value, in addition to noise and the mean, as shown in the equation below:

$$R(t) = \mu + \rho \times R(t-1) + noise$$

Since we only look at one previous step in time, this is an AR(1) model; the model can be extended to include more lagged values and more phi parameters. Here we show an AR(1), an AR(2), and an AR(3).

If phi is equal to one, then the series will represent a random walk as discussed in the previous section, and if it is zero, it will be white noise. In order for the process to be stable and stationary, phi has to be between -1 and +1.

If phi has a negative value, then a positive return last period, at time t-1, implies that this period's return is more likely to be negative. We referred to this as mean reversion in section 1.3.
If phi has a positive value, then a positive return last period implies that this period's return is expected to be positive. We referred to this as momentum in section 1.3.

The example below shows four simulated time series with different phi values (0.9, -0.9, 0.5,-0.5):

```
# Import the module for simulating data
from statsmodels.tsa.arima_process import ArmaProcess

fig, axs = plt.subplots(4)
fig.set_size_inches(12, 14.5)
fig.suptitle('Simulated data with different AR parameters')

# Plot 1: AR parameter = +0.9
ar1 = np.array([1, -0.9])
ma1 = np.array([1])
AR_object1 = ArmaProcess(ar1, ma1)
simulated_data_1 = AR_object1.generate_sample(nsample=1000)
axs[0].plot(simulated_data_1)
axs[0].set_title('Simulated data with Phi = +0.9 ')

# Plot 2: AR parameter = -0.9
ar2 = np.array([1, 0.9])
ma2 = np.array([1])
AR_object2 = ArmaProcess(ar2, ma2)
simulated_data_2 = AR_object2.generate_sample(nsample=1000)
axs[1].plot(simulated_data_2)
axs[1].set_title('Simulated data with Phi = -0.9 ')

# Plot 3: AR parameter = +0.5
ar3 = np.array([1, -0.5])
ma3 = np.array([1])
AR_object3 = ArmaProcess(ar3, ma3)
simulated_data_3 = AR_object3.generate_sample(nsample=1000)
axs[2].plot(simulated_data_3)
axs[2].set_title('Simulated data with Phi = +0.5')

# Plot 3: AR parameter = -0.5
ar4 = np.array([1, 0.5])
ma4 = np.array([1])
```

```
AR_object4 = ArmaProcess(ar4, ma4)
simulated_data_4 = AR_object4.generate_sample(nsample=1000)
axs[3].plot(simulated_data_4)
axs[3].set_title('Simulated data with Phi = -0.5')
```

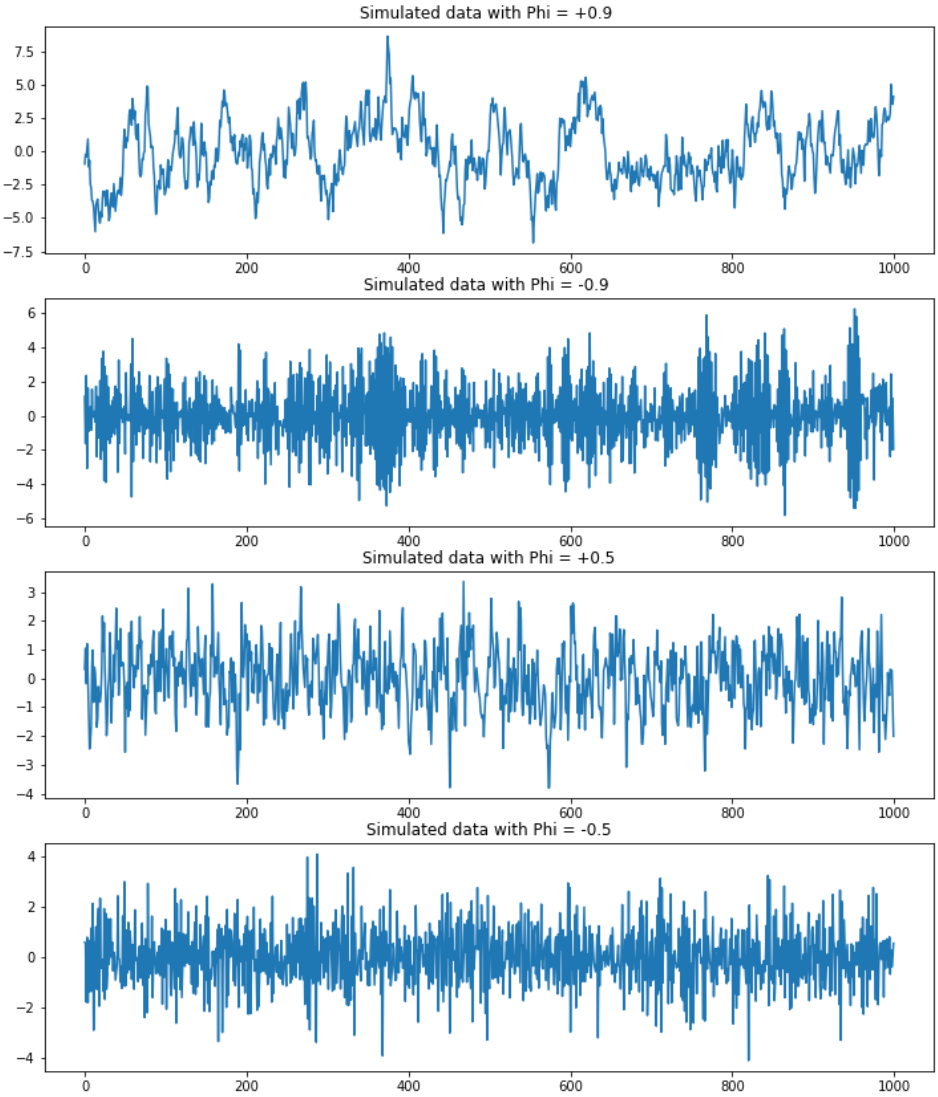


Figure 2.9. Simulated data with different φ values.

When φ is 0.9, it appears to be close to a random walk. When phi equals -0.9, the process has a large positive value that is usually followed by a large negative one. The bottom two are similar but are less exaggerated and closer to white noise.

To have a better understanding, let's look at the autocorrelation function of these four simulated time series:

```
# Import the plot_acf module from statsmodels
from statsmodels.graphics.tsaplots import plot_acf

fig, axs = plt.subplots(2,2,figsize=(15,10))
```

```
fig.suptitle('Autocorrelation functions for different AR parameters')

# Plot 1: AR parameter = +0.9
plot_acf(simulated_data_1 , alpha=1, lags=20, ax=axs[0,0],
title='Autocorrelation function for Phi = +0.9')

# Plot 2: AR parameter = -0.9
plot_acf(simulated_data_2 , alpha=1, lags=20, ax=axs[0,1],
title='Autocorrelation function for Phi = -0.9')

# Plot 3: AR parameter = +0.5
plot_acf(simulated_data_3, alpha=1, lags=20,  ax=axs[1,0],
title='Autocorrelation function for Phi = +0.5')

# Plot 4: AR parameter = -0.5
plot_acf(simulated_data_4, alpha=1, lags=20,  ax=axs[1,1],
title='Autocorrelation function for Phi = -0.5')
```

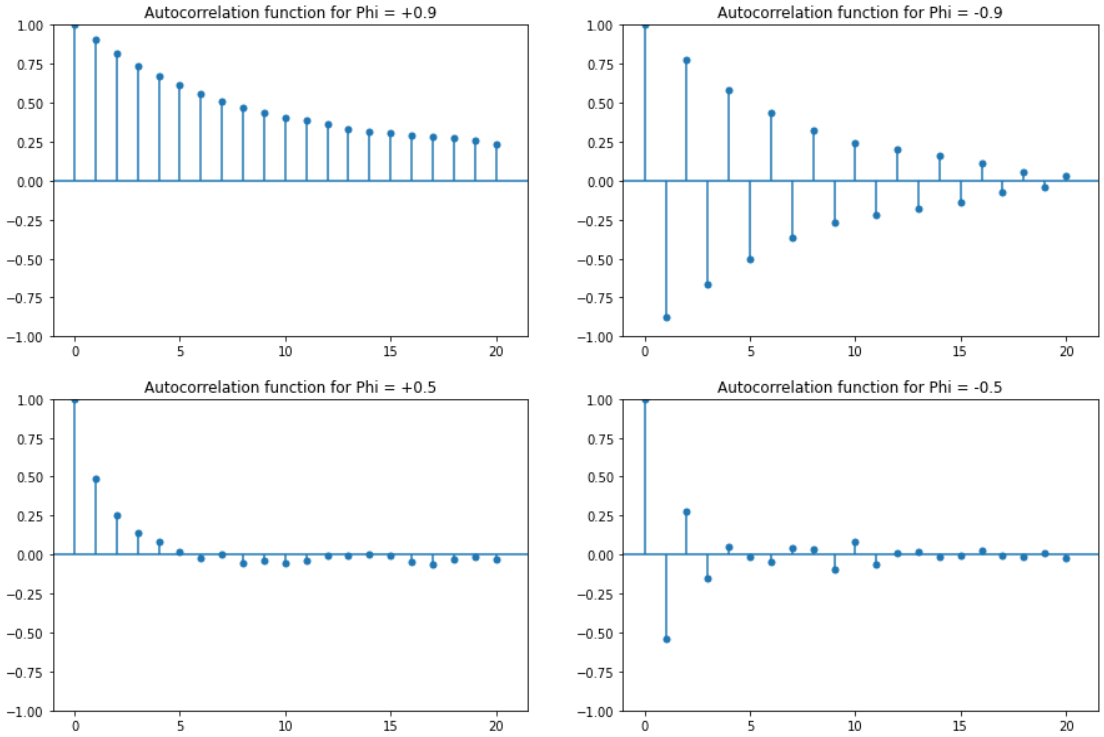


Figure 2.10. Autocorrelation functions for different AR parameters.

If the phi is positive, the autocorrelation function will decay exponentially at the rate of phi. This means that if phi is 0.9, then the autocorrelation at 1 is 0.9, at 2 is 0.9^2 0.81, and at 3 is $0.9^3$0.73, and so on. If phi is negative, then it will be the same, but it will reverse its sign at each lag.

3.2. Estimating & Forecasting AR Models

To estimate the parameters of AR models for a time series, we can use the ARIMA class as shown in the example below:

```
from statsmodels.tsa.arima.model import ARIMA
mod = ARIMA(simulated_data, order=(1,0,0))
result = mod.fit()
print(result.summary())
```

The order (1,0,0) means you're fitting the data to an AR(1) model. An order (2,0,0) would mean you're fitting the data to an AR(2) model. The second and third parts of the order will be discussed in the next section.

The summary of the results is shown below; the phi parameter is highlighted in yellow. We can see that it is 0.91, which is similar to the phi parameter of the simulated data used in the previous subsection.

```
                               SARIMAX Results
==============================================================================
Dep. Variable:                      y   No. Observations:                 1000
Model:                 ARIMA(1, 0, 0)   Log Likelihood               -1437.593
Date:                Mon, 28 Mar 2022   AIC                           2881.185
Time:                        14:36:43   BIC                           2895.909
Sample:                             0   HQIC                          2886.781
                                - 1000
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.2823      0.367      0.769      0.442      -0.437       1.002
ar.L1          0.9123      0.013     72.100      0.000       0.887       0.937
sigma2         1.0361      0.045     22.784      0.000       0.947       1.125
===================================================================================
Ljung-Box (L1) (Q):                   1.11   Jarque-Bera (JB):                 0.48
Prob(Q):                              0.29   Prob(JB):                         0.79
Heteroskedasticity (H):               0.94   Skew:                            -0.04
Prob(H) (two-sided):                  0.58   Kurtosis:                         3.08
===================================================================================
```

3.3. Choosing the Right Model

In the previous subsection, the order of the model was already known. However, in practice, this will not be given to you. There are different methods to determine the order of the AR model. We will focus on two of them: The Partial Autocorrelation Function and the Information Criteria.

The partial autocorrelation function (PACF) measures the incremental benefits of adding another lag. To have a better understanding of PACF, let's first define the partial autocorrelation.
A partial autocorrelation is a description of the relationship between an observation in a time series and data from earlier time steps that **do not take into account the correlations between the intervening observations.**

The correlation between observations at successive time steps is a linear function of the indirect correlations. These indirect connections are eliminated using the partial autocorrelation function. Based on this definition of partial autocorrelation, the PACF indicates only the association between two data points that the shorter lags between those observations do not explain.

The partial autocorrelation for lag 3 is, for example, merely the correlation that lags 1 and 2 do not explain. **In other words, the partial correlation for each lag is the unique correlation between the two observations after the intermediate correlations have been removed**.
To plot the PACF, you can use the plot_pacf function from the statsmodels library as shown in the example below:

```
# Import the modules for simulating data and for plotting the PACF
from statsmodels.tsa.arima_process import ArmaProcess
from statsmodels.graphics.tsaplots import plot_pacf

# Simulate AR(1) with phi=+0.6
ma = np.array([1])
ar = np.array([1, -0.6])
AR_object = ArmaProcess(ar, ma)
simulated_data_1 = AR_object.generate_sample(nsample=5000)

# Plot PACF for AR(1)
plot_pacf(simulated_data_1, lags=20)
plt.show()

# Simulate AR(2) with phi1=+0.6, phi2=+0.3
ma = np.array([1])
ar = np.array([1, -0.6, -0.3])
AR_object = ArmaProcess(ar, ma)
simulated_data_2 = AR_object.generate_sample(nsample=5000)

# Plot PACF for AR(2)
plot_pacf(simulated_data_2, lags=20)
plt.show()
```

We generated two simulated datasets with AR(1) and AR(2). The figure below shows the PACF for both of them:

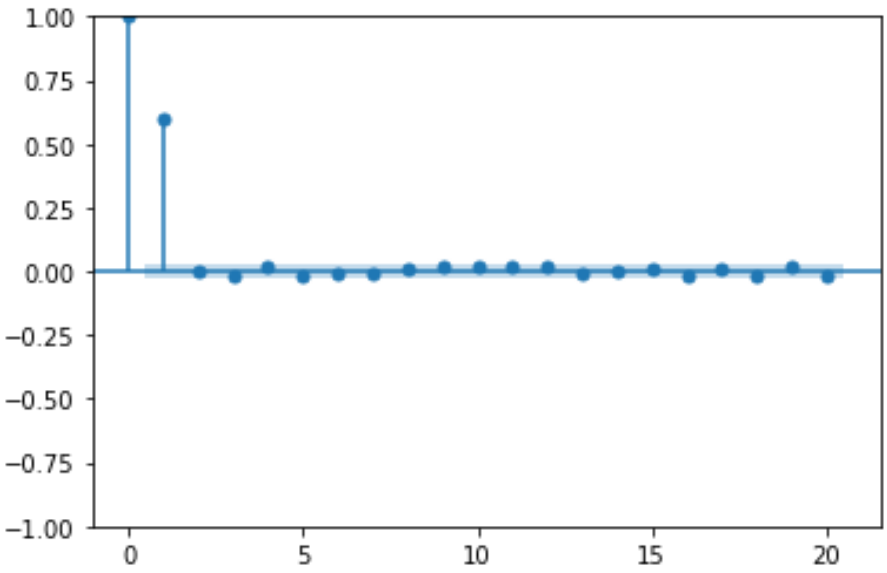


Figure 2.11. PACF for the AR(1) model.

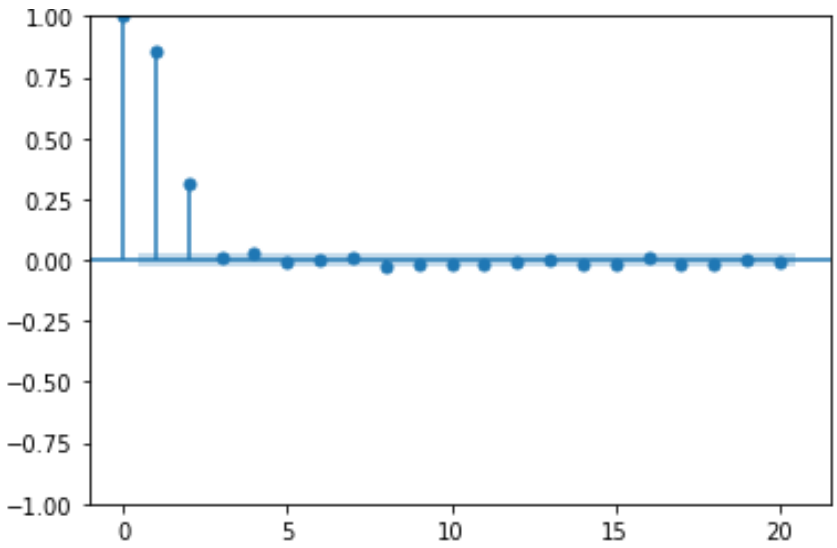


Figure 2.12. PACF for the AR (2) model.

These plots show the Partial Autocorrelation Function for AR models of different orders. In the upper plot, for an AR(1) model, only the lag (1) PACF is significantly different from zero. Similarly, for an AR(2) model, two lags are different from zero.

If we applied the ARIMA function that we used in the previous subsection to the simulated data used in the previous example, and used the order of the model we got from the PACF plot. We should get the same parameters from the function as those used in generating the simulated data. In the example below, we used the simulated data with AR(2) and phi of 0.6 and 0.3:

```
mod = ARIMA(simulated_data_2, order=(2,0,0))
result = mod.fit()
print(result.summary())
```

```
                               SARIMAX Results
==============================================================================
Dep. Variable:                      y   No. Observations:                 5000
Model:                 ARIMA(2, 0, 0)   Log Likelihood               -7085.921
Date:                Wed, 30 Mar 2022   AIC                          14179.841
Time:                        00:48:40   BIC                          14205.910
Sample:                             0   HQIC                         14188.978
                                - 5000
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -0.1471      0.149     -0.990      0.322      -0.438       0.144
ar.L1          0.5885      0.013     44.315      0.000       0.562       0.615
ar.L2          0.3165      0.013     23.809      0.000       0.290       0.343
sigma2         0.9962      0.020     49.383      0.000       0.957       1.036
===================================================================================
Ljung-Box (L1) (Q):                   0.02   Jarque-Bera (JB):                 1.01
Prob(Q):                              0.88   Prob(JB):                         0.60
Heteroskedasticity (H):               0.96   Skew:                             0.03
Prob(H) (two-sided):                  0.44   Kurtosis:                         2.96
===================================================================================
```

The coefficients highlighted in yellow are phi_1 and phi_2, which are, as expected, 0.6 and 0.3. The more parameters in a model, the better the model will fit the data. But this can lead the model to overfit the data. The **information criteria** adjust the goodness-of-fit of a model by imposing a penalty based on the number of parameters used. Two common adjusted goodness-of-fit measures are called the Akaike Information Criterion (AIC) and the Bayesian Information Criterion(BIC).

The AIC is a mathematical method for evaluating how well a model fits the data it was generated. In statistics, AIC is used to compare different possible models and determine which one is the best fit for the data.

The best-fit model according to AIC is the one that explains the greatest amount of variation using the fewest possible independent variables. BIC is a criterion for model selection among a finite set of models.

It is based, in part, on the likelihood function, and it is closely related to AIC. When fitting models, it is possible to increase the likelihood by adding parameters, but doing so may result in overfitting.

The BIC resolves this problem by introducing a penalty term for the number of parameters in the model. The penalty term is larger in BIC than in AIC.
In practice, the way to use the information criteria is to fit several models, each with a different number of parameters, and choose the one with the lowest Bayesian information criterion. This is shown in the example below:

```
# Import the module for estimating an ARMA model
from statsmodels.tsa.arima.model import ARIMA

# Fit the data to an AR(p) for p = 0,...,6 , and save the BIC
BIC = np.zeros(7)
for p in range(7):
    mod = ARIMA(simulated_data_2, order=(p,0,0))
    res = mod.fit()
# Save BIC for AR(p)
    BIC[p] = res.bic

# Plot the BIC as a function of p
plt.plot(range(1,7), BIC[1:7], marker='o')
plt.xlabel('Order of AR Model')
plt.ylabel('Bayesian Information Criterion')
```

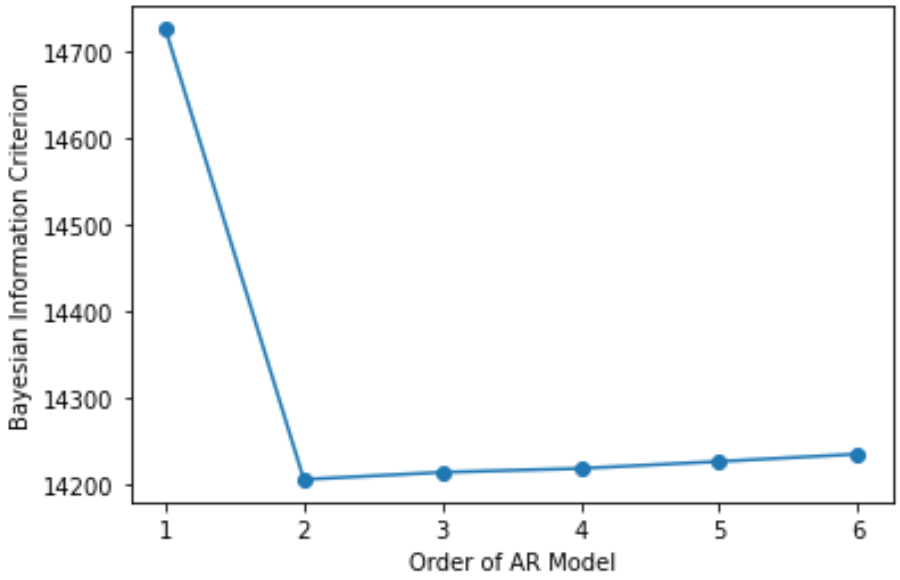


Figure 2.13. BIC for the simulated data from the AR (2) model.

You can see that the lowest BIC occurs for an AR(2), which is what the simulated data was from.

4. Moving Average and ARMA Models

In this section, we will go through another kind of model, the moving average, or MA, model. We will also see how to combine AR and MA models into a powerful ARMA model.

4.1.Moving Average Model Definition

In the MA model, today's values equal a mean plus noise, plus a fraction θ of yesterday's noise. This is shown in the following equation:

$$R(t) = \mu + noise + \theta \times (yesterday - noise)$$

This is called an MA model of order 1, or simply an MA(1) model, as we take into consideration only yesterday's noise. If we look for the two previous days, it will be MA(2), and so on.
If the MA parameter, θ, is zero, then the process is white noise. MA models are stationary for all values of θ.

Suppose R(t) is a time series of stock returns. If θ is negative, then a positive shock last period, represented by epsilon(t-1), would have caused the last period's return to be positive, but this period's return is more likely to be negative. A shock two periods ago would not affect today's return—only the shock now and last period.

To create simulated data from the MA model, we can use ArmaProcess from the statsmodels library as shown in the example below:

```
from statsmodels.tsa.arima_process import ArmaProcess
ar = np.array([1])
ma = np.array([1, 0.5])
AR_object = ArmaProcess(ar, ma)
simulated_data = AR_object.generate_sample(nsample=1000)
plt.plot(simulated_data)
```

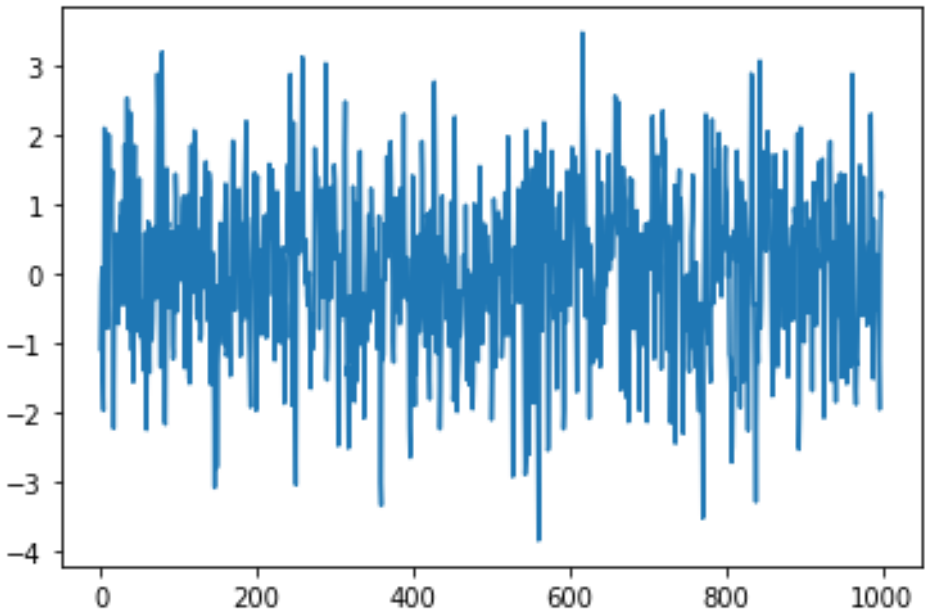


Figure 2.14. Simulated data from the MA (1) model.

4.2. Estimation of the MA Model

To estimate the MA model from a given time series, we can use the same method as in estimating the AR model shown before in section 3.2. We will use the ARIMA function, but the order will be (0,0,1) instead of (1,0,0). An example of this is shown below:

```
from statsmodels.tsa.arima.model import ARIMA
mod = ARIMA(simulated_data, order=(0,0,1))
result = mod.fit()
```

```
print(result.summary())
```

```
                               SARIMAX Results
==============================================================================
Dep. Variable:                      y   No. Observations:                 1000
Model:                 ARIMA(0, 0, 1)   Log Likelihood               -1445.804
Date:                Wed, 30 Mar 2022   AIC                           2897.608
Time:                        19:47:35   BIC                           2912.331
Sample:                             0   HQIC                          2903.204
                                - 1000
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const         -0.0300      0.047     -0.634      0.526      -0.123       0.063
ma.L1          0.4543      0.029     15.745      0.000       0.398       0.511
sigma2         1.0549      0.051     20.820      0.000       0.956       1.154
===================================================================================
Ljung-Box (L1) (Q):                   0.03   Jarque-Bera (JB):                 3.26
Prob(Q):                              0.87   Prob(JB):                         0.20
Heteroskedasticity (H):               0.87   Skew:                             0.05
Prob(H) (two-sided):                  0.22   Kurtosis:                         2.74
===================================================================================
```

The number highlighted in yellow is the θ parameter for the simulated data. The result agrees with our expectation, as the parameter used for generating the simulation data was 0.

4.3. ARMA Models

An ARMA model is a combination of both the AR and MA models. Here is the formula of the ARMA(1,1) model:

$$R(t) = \mu + \theta \times Noise(t-1) + \pi \times R(t-1) + Noise(t)$$

5. Case Study: Climate Change

Let's put all the concepts covered in this article together through the climate change case study. We will analyze some temperature data taken over almost 150 years. The data was downloaded from the **NOAA website**.

The following steps will be applied:

- Apply pandas methods by converting the index to DateTime and plotting the data.
- Apply the Augmented Dickey-Fuller test to see whether the data is a Random Walk.
- Take the first differences in the data to transform it into a stationary series
- Compute the Autocorrelation Function and the Partial Autocorrelation Function of the data.
- Fit AR, MA, and ARMA models to the data.
- Use the Information Criterion to choose the best model among the ones you looked at.
- Finally, with the best model, forecast temperatures over the next 30 years.

The first two steps are done with the code below:

```
# Import the adfuller function from the statsmodels module
from statsmodels.tsa.stattools import adfuller

#upload the data
temp_NY = pd.read_csv('New York temperature data.csv',
parse_dates=['DATE'], index_col='DATE' )

# Convert the index to a datetime object
temp_NY.index = pd.to_datetime(temp_NY.index, format='%Y')

# Plot average temperatures
temp_NY.plot()
plt.show()

# Compute and print ADF p-value
result = adfuller(temp_NY['AVG'])
print("The p-value for the ADF test is ", result[1])
```

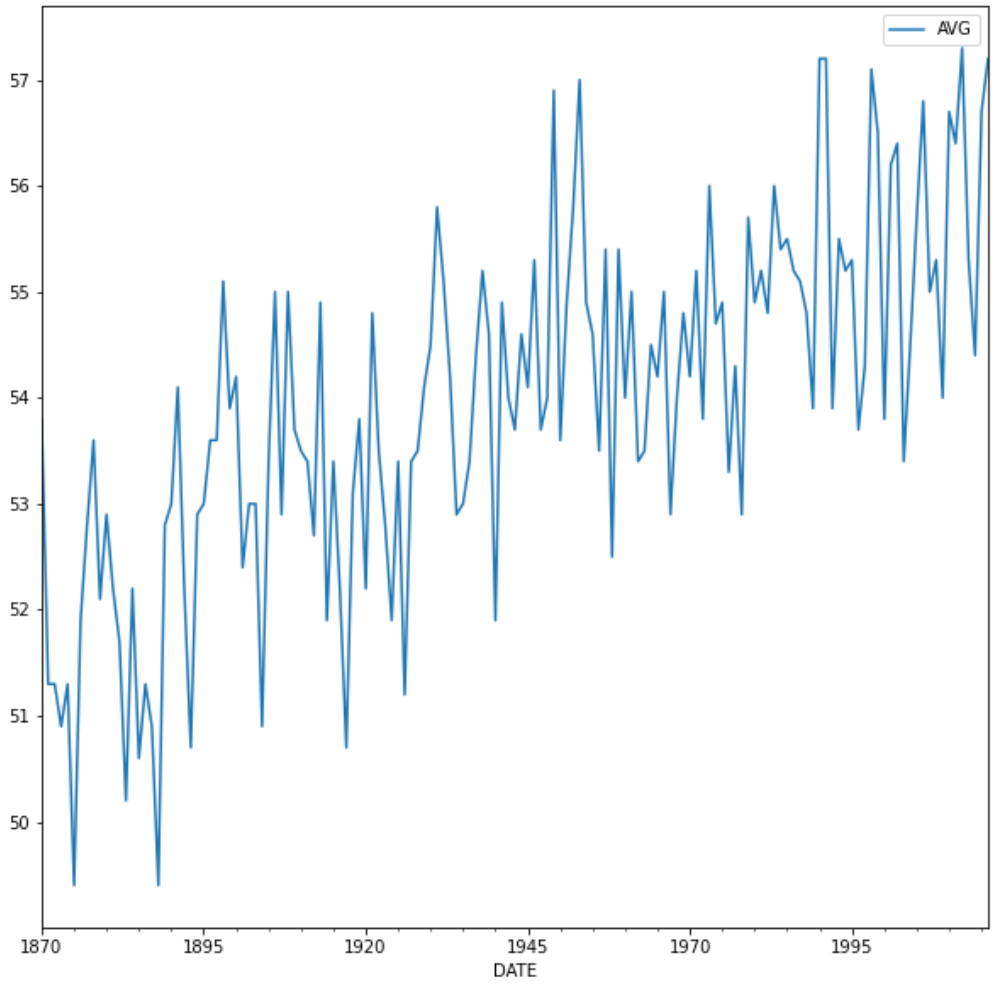


Figure 2.15. The average temp of New York City.

The p-value of the ADF test is 0.58, which means that the time series is not stationary and we cannot reject that it is a random walk. The third and fourth steps are applied to the data using the code below:

```
# Import the modules for plotting the sample ACF and PACF
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# Take first difference of the temperature Series
chg_temp = temp_NY.diff()
chg_temp = chg_temp.dropna()

# Plot the ACF and PACF on the same page
fig, axes = plt.subplots(2,1)

# Plot the ACF
plot_acf(chg_temp, lags=20, ax=axes[0])

# Plot the PACF
plot_pacf(chg_temp, lags=20, ax=axes[1])
plt.show()
```

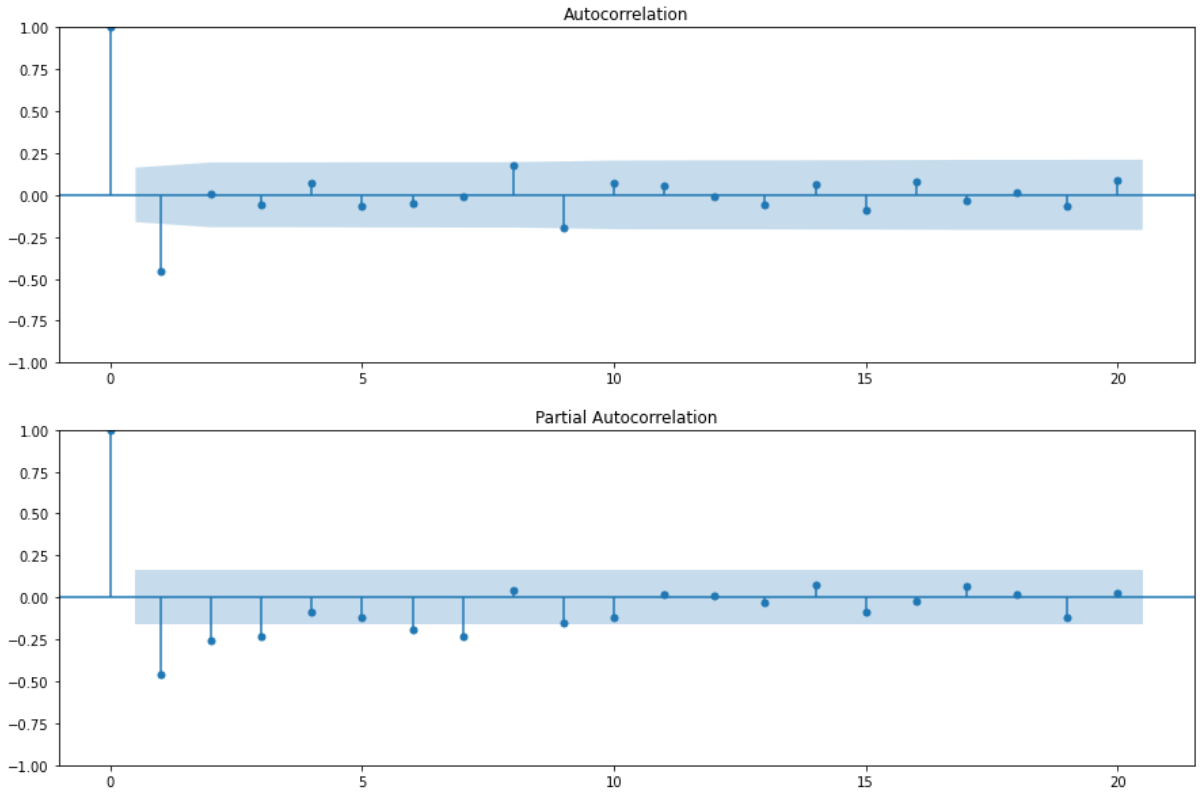


Figure 2.16. Correlation and autocorrelation function of the first difference of the data.

There is no clear pattern in the ACF and PACF except for the negative lag-1 autocorrelation in the ACF. After that, the three data models will be fitted to the data, and the AIC is calculated for each model. This is done using the code below:

```
# Import the module for estimating an ARIMA model
from statsmodels.tsa.arima.model import ARIMA

# Fit the data to an AR(1) model and print AIC:
mod_ar1 = ARIMA(chg_temp, order=(1,0,0))
res_ar1 = mod_ar1.fit()
print("The AIC for an AR(1) is: ", res_ar1.aic)

# Fit the data to an AR(2) model and print AIC:
mod_ar2 = ARIMA(chg_temp, order=(2,0,0))
res_ar2 = mod_ar2.fit()
print("The AIC for an AR(2) is: ", res_ar2.aic)

# Fit the data to an ARMA(1,1) model and print AIC:
mod_arma11 = ARIMA(chg_temp, order=(1,0,1))
res_arma11 = mod_arma11.fit()
print("The AIC for an ARMA(1,1) is: ", res_arma11.aic)
```

```
The AIC for an AR(1) is:  510.5346898733109
The AIC for an AR(2) is:  501.92741234091363
The AIC for an ARMA(1,1) is:  469.0729194692342
```

The ARMA(1,1) has the lowest AIC values among the three models. Therefore, it will be used for parameter estimation and forecasting.

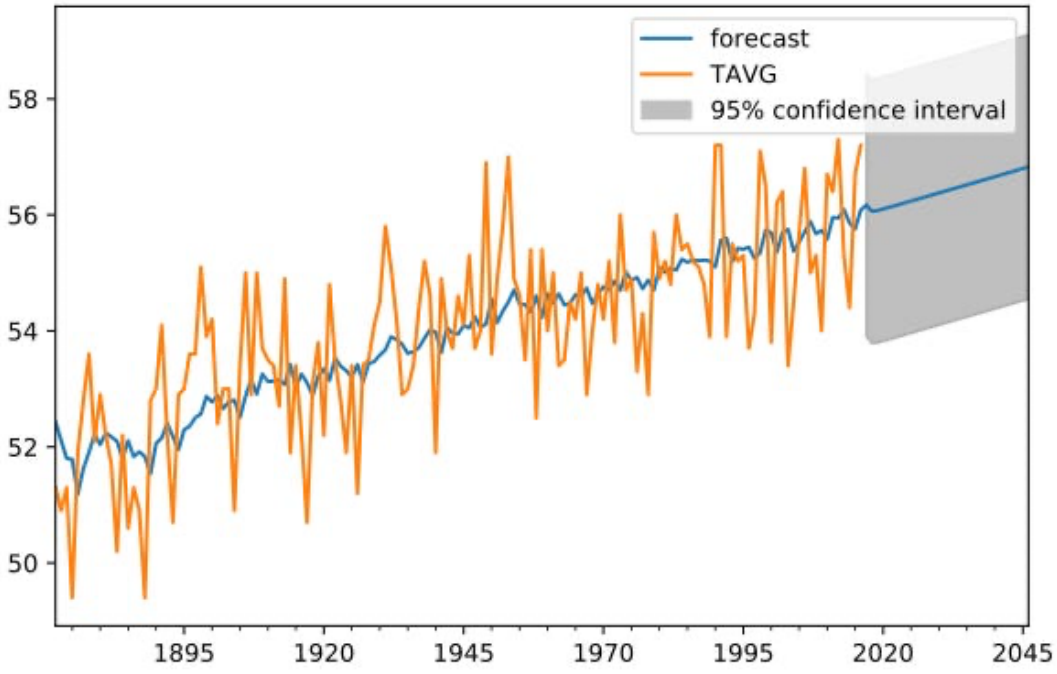


Figure 17. Forecast of Average Temperatures using an ARMA(1,1) Model

According to the model, the temperature is expected to be about 0.6 degrees higher in 30 years (almost entirely due to the trend), but the 95% confidence interval around that is over 5 degrees.

Visualizing Time Series Data

Table of Contents:

3

1. Line Plots

In this section, we will learn how to leverage basic plotting tools in Python and how to annotate and personalize your time series plots.

1.1. Create time-series line plots

First, we will upload the discoveries dataset and set the date as the index using .**read_csv,** and then the plot will use the **.plot** method as shown in the code below:

```
df = pd.read_csv('discoveries.csv', parse_dates=['date'],
index_col='date')
df.plot(figsize=(10,10))
plt.show()
```

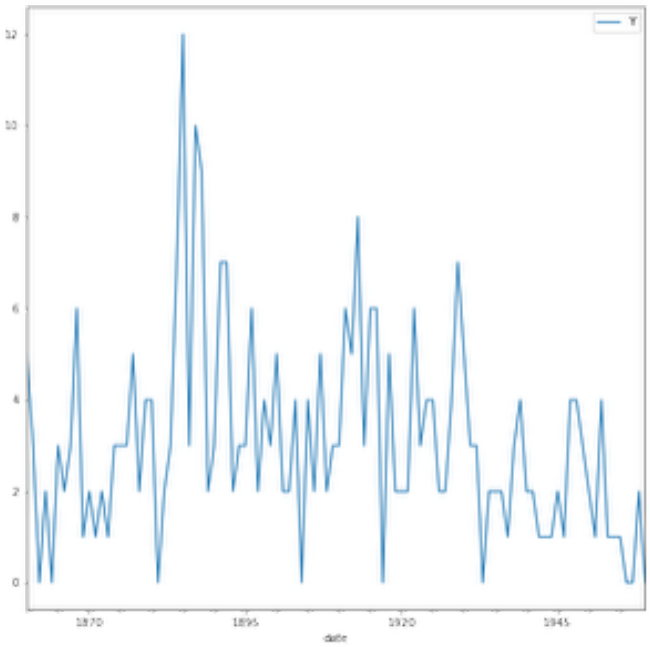


Figure 3.1. The great inventions and scientific discoveries from 1860 to 1959.

The default style for the **matplotlib** plot may not necessarily be your preferred style, but it is possible to change that. Because it would be time-consuming to customize each plot or to create your own template, several matplotlib style templates have been made available for use.

These can be invoked by using the **plt.style** command, and will automatically add pre-specified defaults for fonts, lines, points, background colors, etc, to your plots. In this case, we opted to use the famous **fivethirtyeight** style sheet. To set this style, you can use the code below:

```
plt.style.use('fivethirtyeight')
df.plot(figsize=(10,10))
plt.show()
```

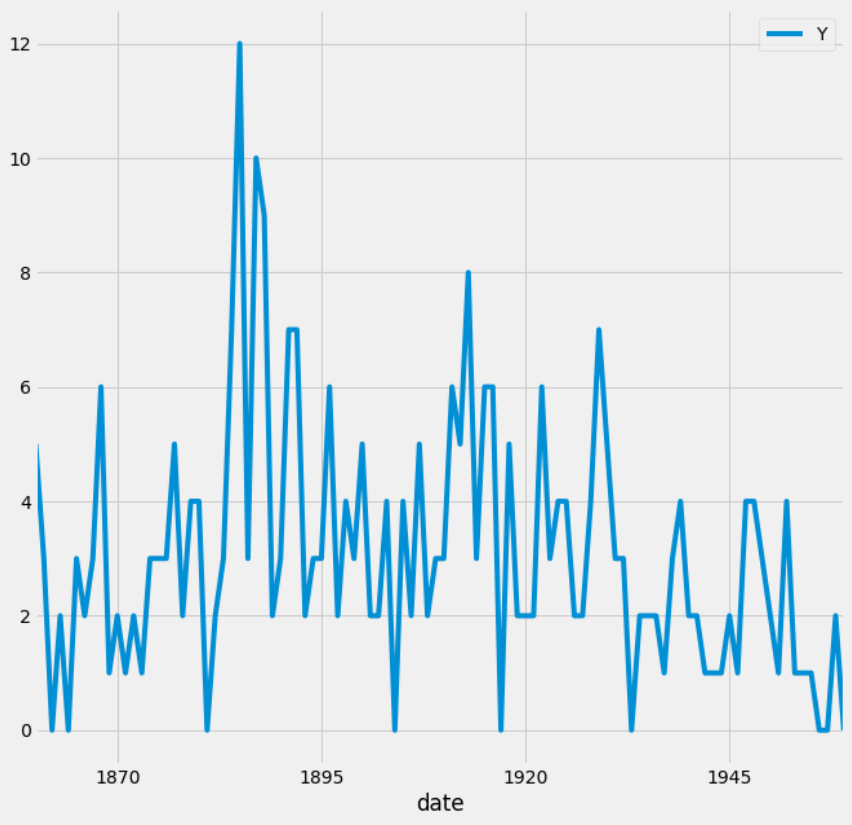


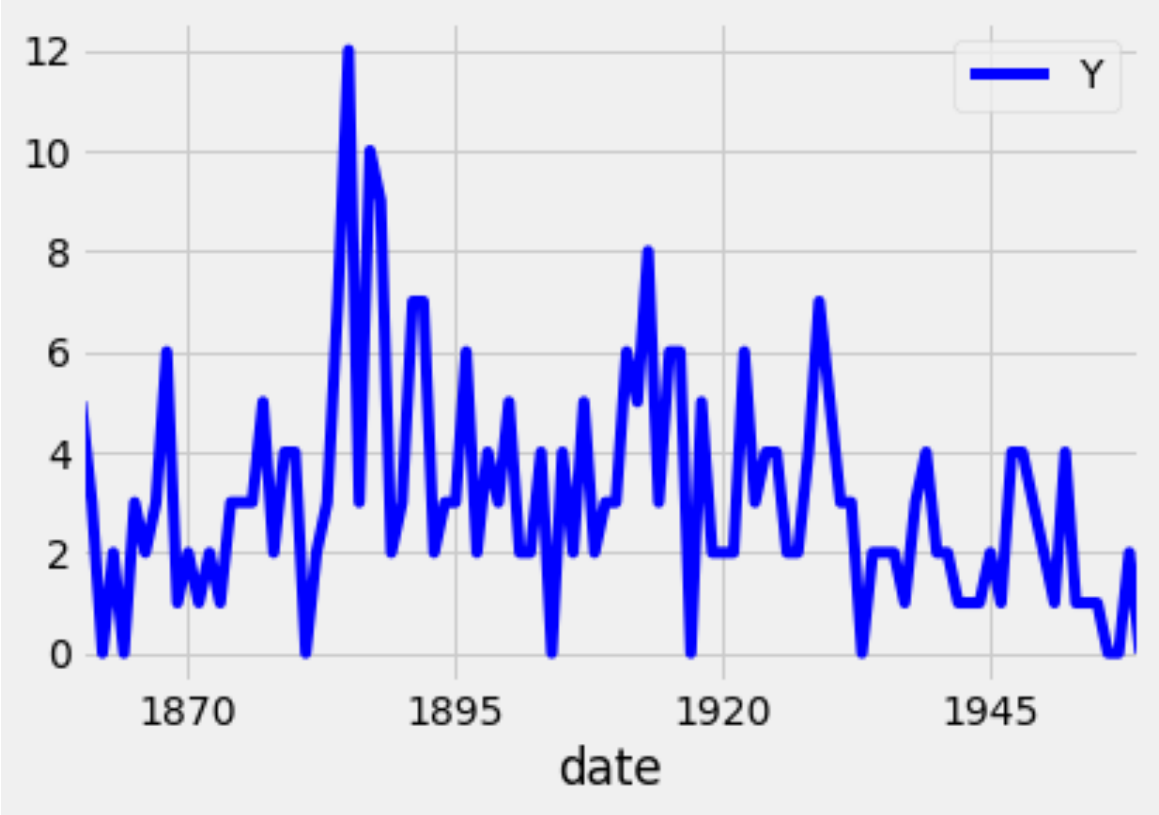
Figure 3.2. The great inventions and scientific discoveries from 1860 to 1959 used the FiveThirtyEight style.

To see all of the available styles, use the following code:

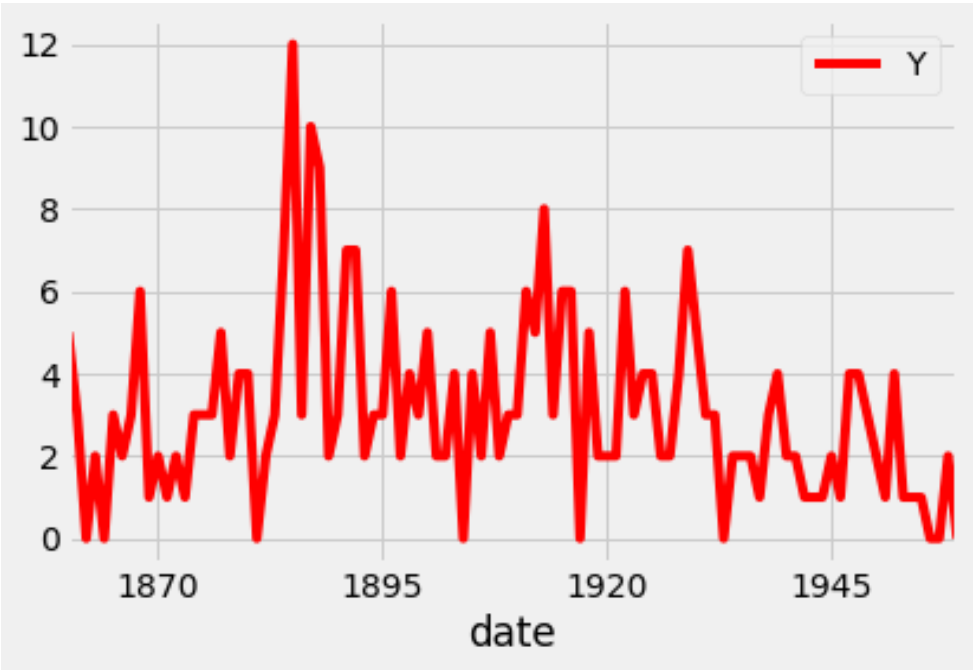
```
print(plt.style.available)
```

You can also change the color of the plot using the color parameter, as shown in the code below:

```
ax = df.plot(color='blue')
```




```
ax = df.plot(color='red')
```



Since your plots should always tell a story and communicate the relevant information. Therefore, each of your plots must be carefully annotated with axis labels and legends.

The **.plot()** method in pandas returns a matplotlib AxesSubplot object, and it is common practice to assign this returned object to a variable called ax. Doing so also allows you to include additional notations and specifications to your plot, such as axis labels and titles.

In particular, you can use the **.set_xlabel()**, **.set_ylabel()**, and **.set_title()** methods to specify the x and y-axis labels and titles of your plot.

```
ax = df.plot(color='blue', figsize=(10,10))
ax.set_xlabel('Date')
ax.set_ylabel('Number of great discoveries')
ax.set_title('Number of great inventions and scientific discoveries
from 1860 to 1959')
plt.show()
```

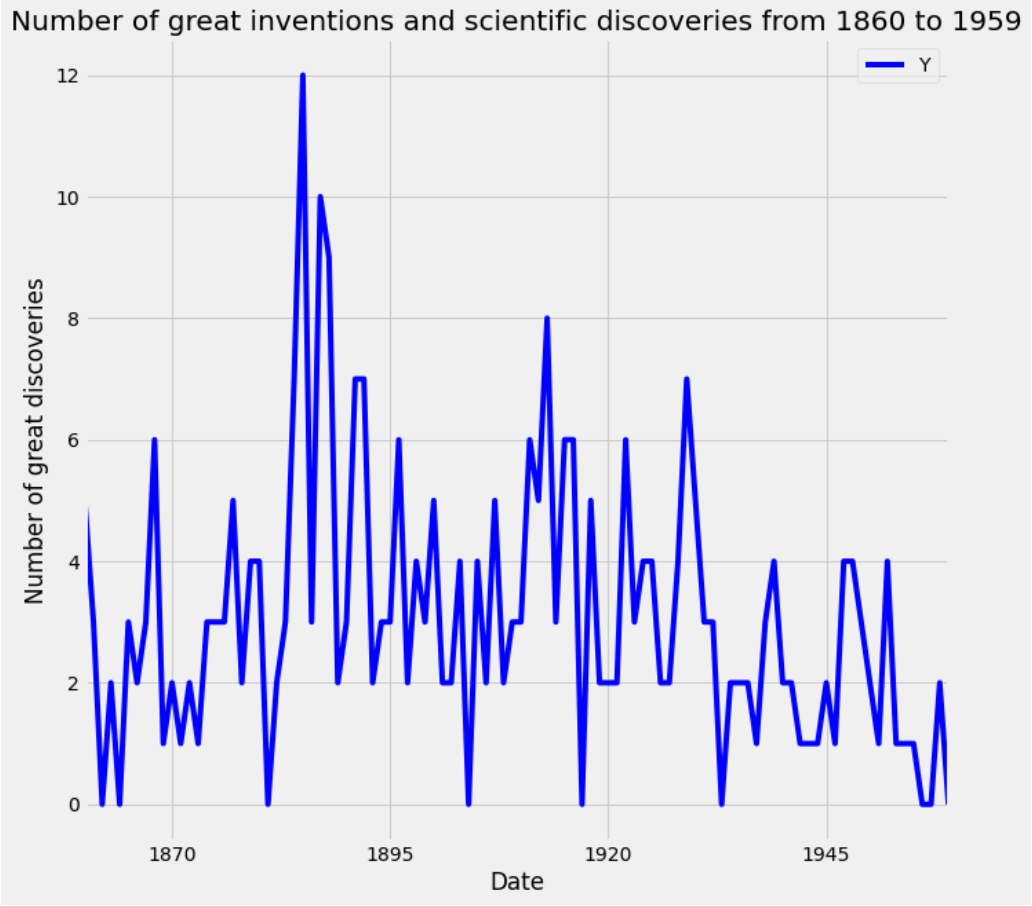


Figure 3.5. The great inventions and scientific discoveries from 1860 to 1959 used the FiveThirtyEight style and with an added title and x-label, and y-label.

1.2. Customize your time series plot

Plots are great because they allow users to understand the data. However, you may sometimes want to highlight specific events or guide the user through your train of thought.

To plot a subset of the data and the data index of the pandas DataFrame consists of dates, you can slice the data using strings that represent the period in which you are interested. This is shown in the example below:

```
df_subset = df['1860':'1870']
ax = df_subset.plot(color='blue', fontsize=14)
plt.show()
```

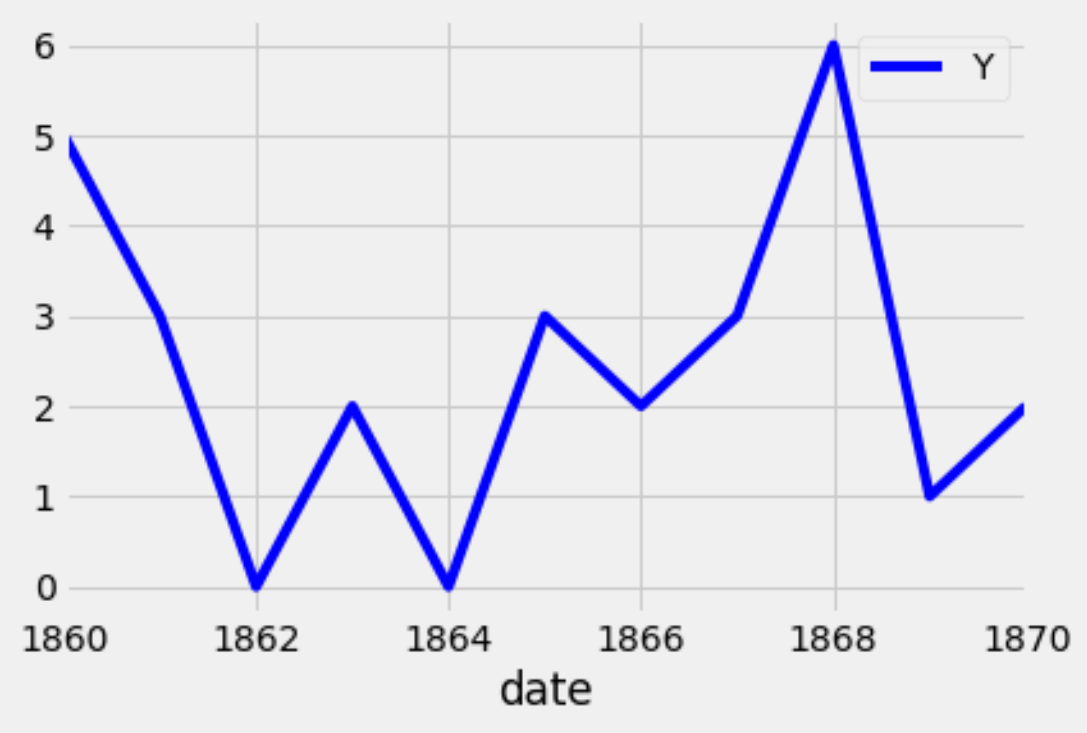


Figure 3.6. The great inventions and scientific discoveries from 1860 to 1870.

Additional annotations can also help emphasize specific observations or events in your time series. This can be achieved with **matplotlib** by using the **axvline** and **axvhline** methods. This is shown in the example below, in which vertical and horizontal lines are drawn using **axvline** and **axvhline** methods.

```
# adding markers
ax = df.plot(color='blue', figsize=(12,10))
ax.set_xlabel('Date')
ax.set_ylabel('Number of great discoveries')
ax.axvline('1920-01-01', color='red', linestyle='--')
ax.axhline(4, color='green', linestyle='--')
```

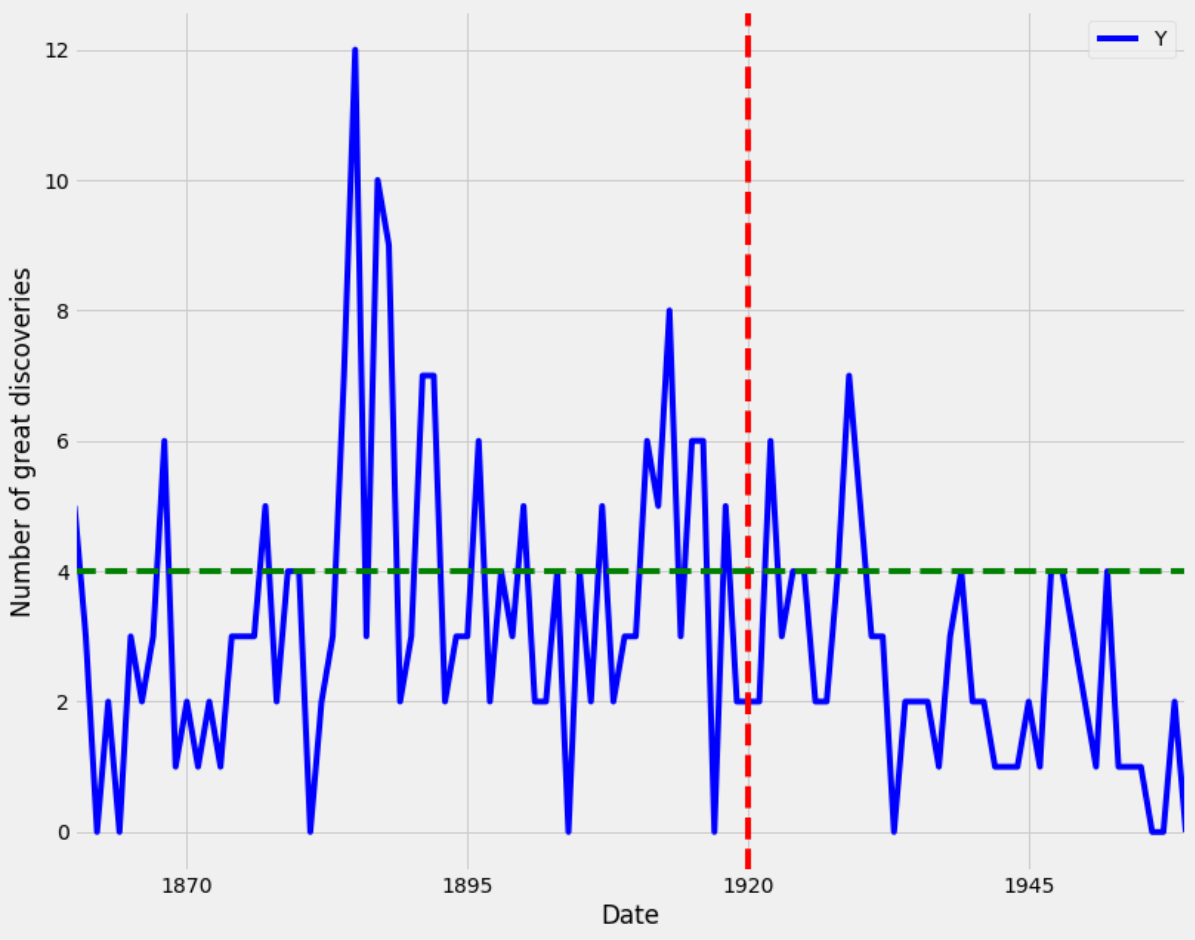


Figure 3.7. The great inventions and scientific discoveries from 1860 to 1950, with vertical lines in 1920 and horizontal lines at 4.

Beyond annotations, you can also highlight regions of interest in your time series plot. This can help provide more context around your data and emphasize the story you are trying to convey with your graphs.

To add a shaded section to a specific region of your plot, you can use the **axvspan** and **axhspan** methods in **matplolib** to produce vertical regions and horizontal regions, respectively. An example of this is shown in the code below:

```
# Highlighting regions of interest
ax = df.plot(color='blue', figsize=(15,10))
ax.set_xlabel('Date')
ax.set_ylabel('Number of great discoveries')
ax.axvspan('1890-01-01', '1910-01-01', color='red', alpha=0.3)
ax.axhspan(8, 6, color='green', alpha=0.3)
```

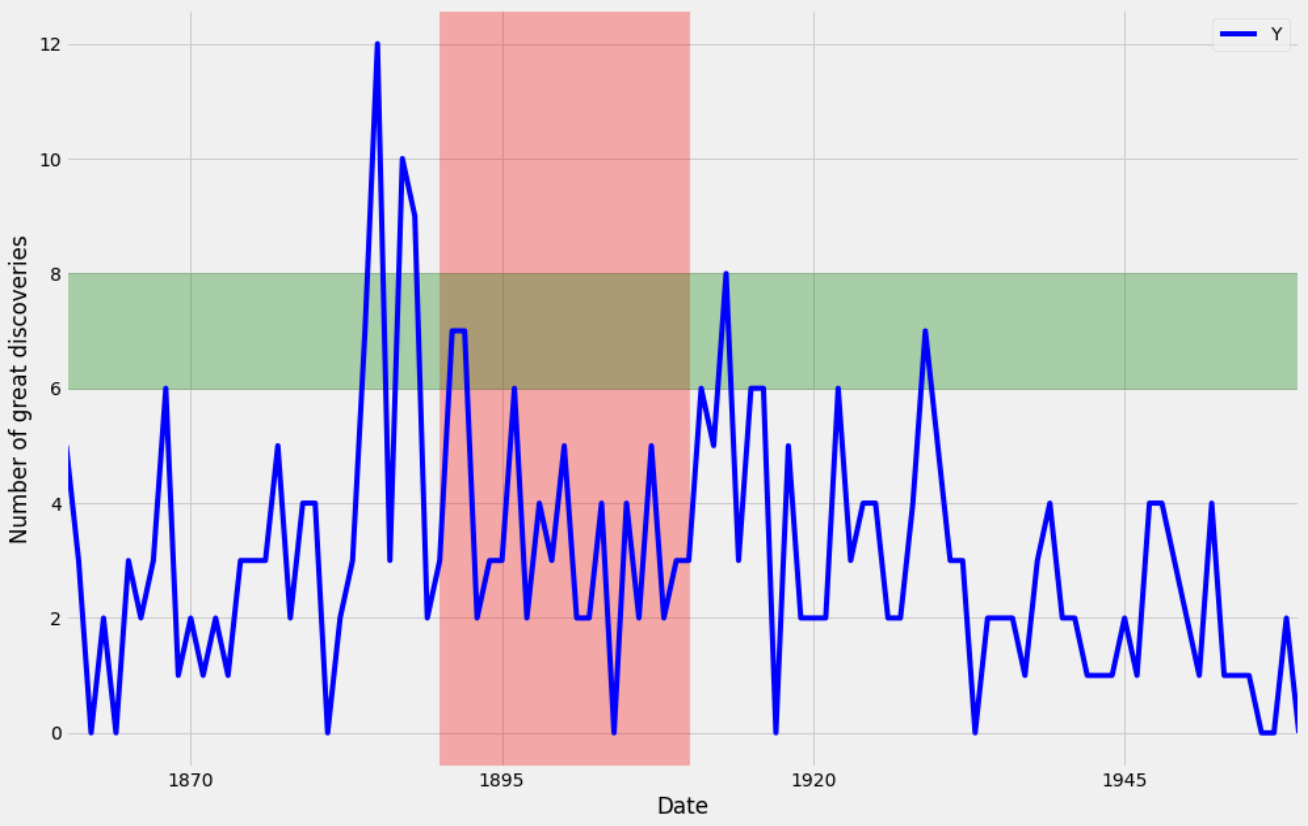


Figure 3.8. The great inventions and scientific discoveries from 1860 to 1950, with vertically highlighted regions from 1890 to 1910 and highlighted horizontal region lines from 6 to 8.

2. Visualizing Summary Statistics and Diagnostics

In this section, we will explain how to gain a deeper understanding of your time-series data by computing summary statistics and plotting aggregated views of your data.

In this section, we will use a new dataset that is famous within the time series community. This time series dataset contains the CO2 measurements at the Mauna Loa Observatory, Hawaii, between the years 1958 and 2001. The dataset can be downloaded from here.

2.1. Clean your time series data

In real-life scenarios, data can often come in messy and/or noisy formats. “Noise” in data can include things such as outliers, misformatted data points, and missing values.
In order to be able to perform an adequate analysis of your data, it is important to carefully process and clean your data. While this may seem like it will slow down your analysis initially, this investment is critical for future development and can really help speed up your investigative analysis.

The first step to achieving this goal is to check your data for missing values. In pandas, missing values in a DataFrame can be found with the **.isnull()** method. Inversely, rows with non-null values can be found with the **.notnull()** method. In both cases, these methods return True/False values where non-missing and missing values are located.

If you are interested in finding how many rows contain missing values, you can combine the **.isnull()** method with the **.sum()** method to count the total number of missing values in each of the columns of the df DataFrame. This works because **df.isnull()** returns the value True if a row value is null, and dot sum() returns the total number of missing rows. This is done with the code below:

```
# count missing values
print(co2_levels.isnull().sum())
```

The number of missing values is 59 rows. To replace the missing values in the data, we can use different options such as the mean value, the value from the preceding time point, or the value from time points that are coming after.

To replace missing values in your time series data, you can use the **.fillna()** method in pandas. It is important to notice the method argument, which specifies how we want to deal with our missing data.

Using the method **bfill** (i.e, backfilling) will ensure that missing values are replaced by the next valid observation. On the other hand, **ffill()** (i.e., forward filling) will replace the missing values with the most recent non-missing value. Here, we will use the **bfill** method.

```
#Replacing missing values in a DataFrame
co2_levels = co2_levels.fillna(method='bfill')
```

2.2. Plot aggregates of your data

A moving average, also known as a rolling mean, is a commonly used technique in the field of time series analysis. It can be used to smooth out short-term fluctuations, remove outliers, and highlight long-term trends or cycles.

Taking the rolling mean of your time series is equivalent to "smoothing" your time series data. In pandas, the **.rolling()** method allows you to specify the number of data points to use when computing your metrics.

Here, you specify a sliding window of 52 points and compute the mean of those 52 points as the window moves along the date axis. The number of points to use when computing moving averages depends on the application, and these parameters are usually set through trial and error or according to some seasonality.

For example, you could take the rolling mean of daily data and specify a window of 7 to obtain weekly moving averages. In our case, we are working with weekly data, so we specified a window of 52 (because there are 52 weeks in a year) in order to capture the yearly rolling mean. The rolling mean of a window of 52 is applied to the data using the code below:

```
# The moving average model
co2_levels_mean = co2_levels.rolling(window=52).mean()
ax = co2_levels_mean.plot(figsize=(12,10))
```

```
ax.set_xlabel("Date")
ax.set_ylabel("The values of my Y axis")
ax.set_title("52 weeks rolling mean of my time series")
plt.show()
```

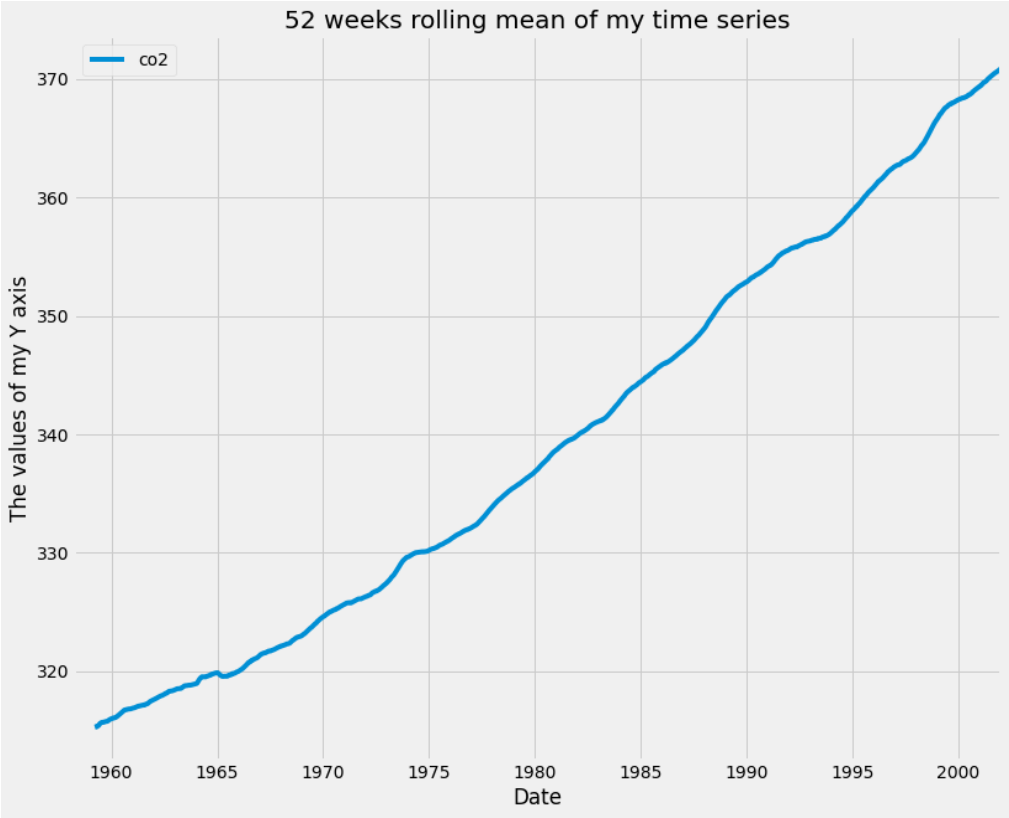


Figure 3.9. 52 weeks rolling mean of the CO2 levels time series.

Another useful technique to visualize time series data is to take aggregates of the values in your data. For example, the CO2_levels data contains weekly data, but you may wish to see how these values behave by month of the year.

Because you have set the index of your CO2_levels DataFrame as a DateTime type, it is possible to directly extract the day, month, or year of each date in the index. For example, you can extract the month using the command co2_levels **.index .month**. Similarly, you can extract the year using the command co2_levels **.index .year**.

Aggregating values in a time series can help answer questions such as "what is the mean value of our time series on Sundays?" or "what is the mean value of our time series during each month of the year?".

If the index of your pandas DataFrame consists of DateTime types, then you can extract the indices and group your data by these values. Here, you use the **.groupby()** and **.mean()** methods

to compute the monthly and yearly averages of the CO2 levels data and assign that to a new variable called co2_levels_by_month and co2_levels_by_year.

The **.groupby()** method allows you to group records into buckets based on a set of defined categories. In this case, the categories are the different months of the year and for each year.

```
# Plotting aggregate values of your time series
index_month = co2_levels.index.month
co2_levels_by_month = co2_levels.groupby(index_month).mean()
co2_levels_by_month.plot(figsize=(12,10))
plt.show()
```

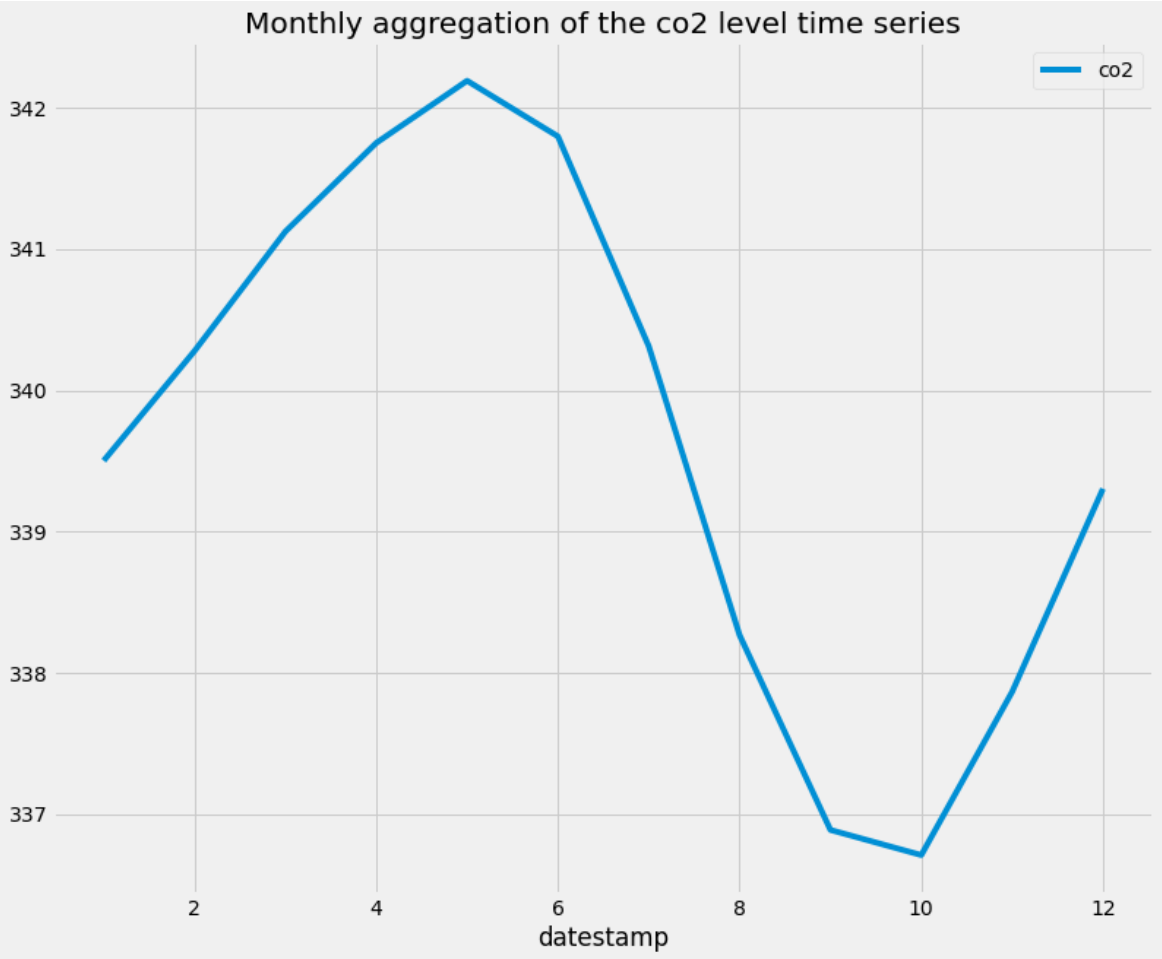


Figure 3.10. Monthly aggregation of the CO2 level time series.

When we plot **co2_levels_by_month**, we see that the monthly mean value of CO2 levels peaks during the 5th to 7th months of the year. This is consistent with the fact that during summer, we see increased sunlight and CO2 emissions from the environment. I like this example, as it shows the power of plotting aggregated values of time series data.

```
# Plotting aggregate values of your time series
index_year = co2_levels.index.year
```

```
co2_levels_by_year = co2_levels.groupby(index_year).mean()
co2_levels_by_year.plot(figsize=(12,10))
plt.title('Yearly aggregation of the co2 level time series')
plt.show()
```

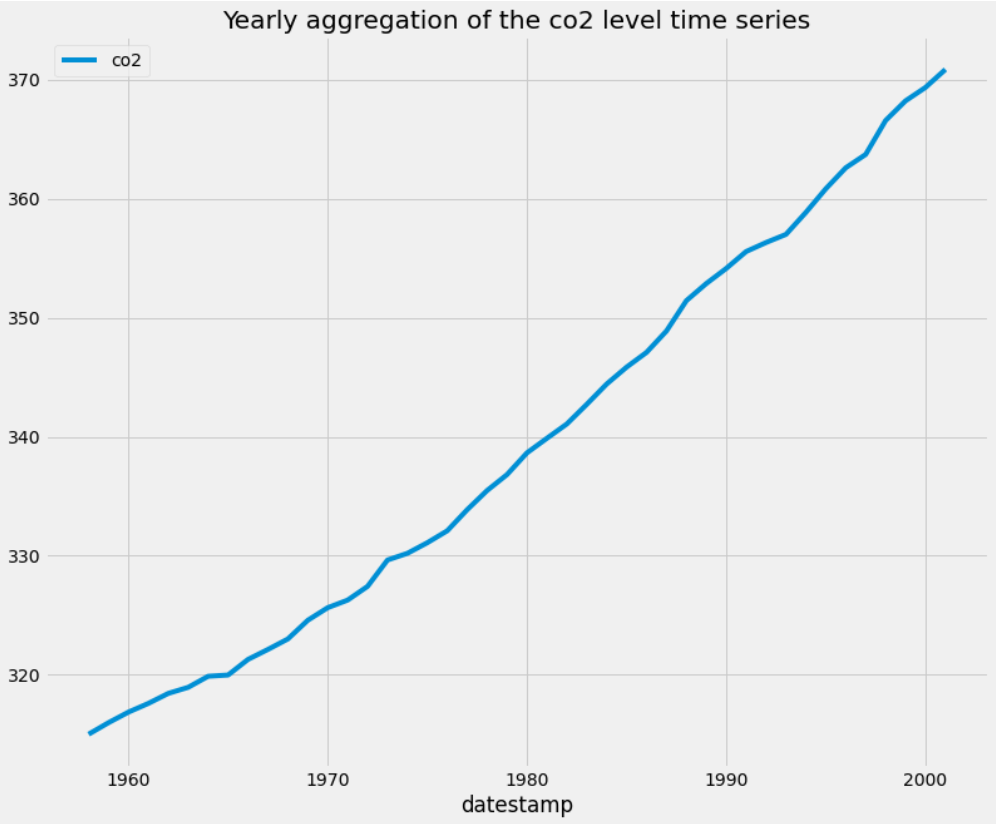


Figure 3.11. Yearly aggregation of the CO2 level time series.

When we plot co2_levels_by_year, we can see that the CO2 level is increasing every year, which is expected.

2.3. Summarize the values in the dataset

An important step to understanding the data is to create summary statistics plots of the time series that you are working with. Doing so will allow you to share and discuss the statistical properties of your data that can further support the plots that you generate and any hypotheses that you want to communicate. There are three fundamental plots to visualize the summary statistics of the data: the box plot, histogram plot, and density plot.

A boxplot provides information on the shape, variability, and median of your data. It is particularly useful to display the range of your data and for identifying any potential outliers. The lines extending parallel from the boxes are commonly referred to as “whiskers”, which are used to indicate variability outside the upper (which is the 75% percentile) and lower (which is

the 25% percentile) quartiles, i.e., outliers. These outliers are usually plotted as individual dots that are in line with whiskers.

```
# Summarizing your data with boxplots
ax1 = co2_levels.boxplot(figsize=(12,10))
ax1.set_ylabel('Co2 levels')
ax1.set_title('Boxplot for the co2 levels data')
```

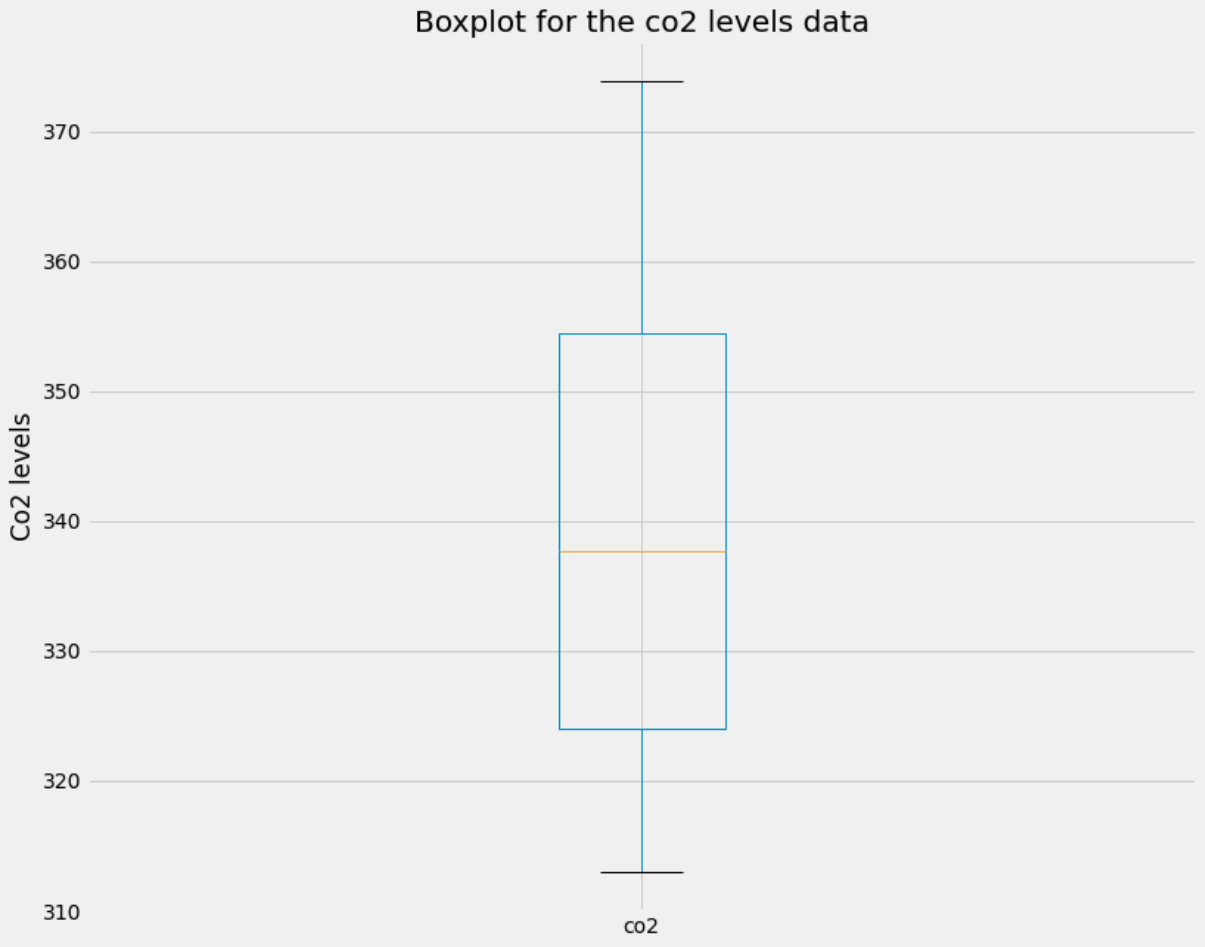


Figure 3.12. Box plot for the CO2 levels data.

Histograms are a type of plot that allows you to inspect the underlying distribution of your data. It visualizes the frequency of occurrence of each value in your data.

These can sometimes be more useful than boxplots, as non-technical members of your team will often be more familiar with histograms, and therefore are more likely to quickly understand the shape of the data you are exploring or presenting to them.

In pandas, it is possible to produce a histogram by simply using the standard **.plot()** method and specifying the kind argument as **hist**. In addition, you can select the bins parameter, which determines how many intervals you should cut your data into.

Regarding the **bin** parameter, there are no hard and fast rules for finding the optimal value; it often needs to be determined through trial and error.

```
# Summarizing your data with density plots
ax3 = co2_levels.plot(kind='density', linewidth=2, figsize=(15,10))
ax3.set_xlabel('co2 levels data values')
ax3.set_ylabel('Density values of the co2 levels data')
ax3.set_title('Density plot of the co2 levels data')
```

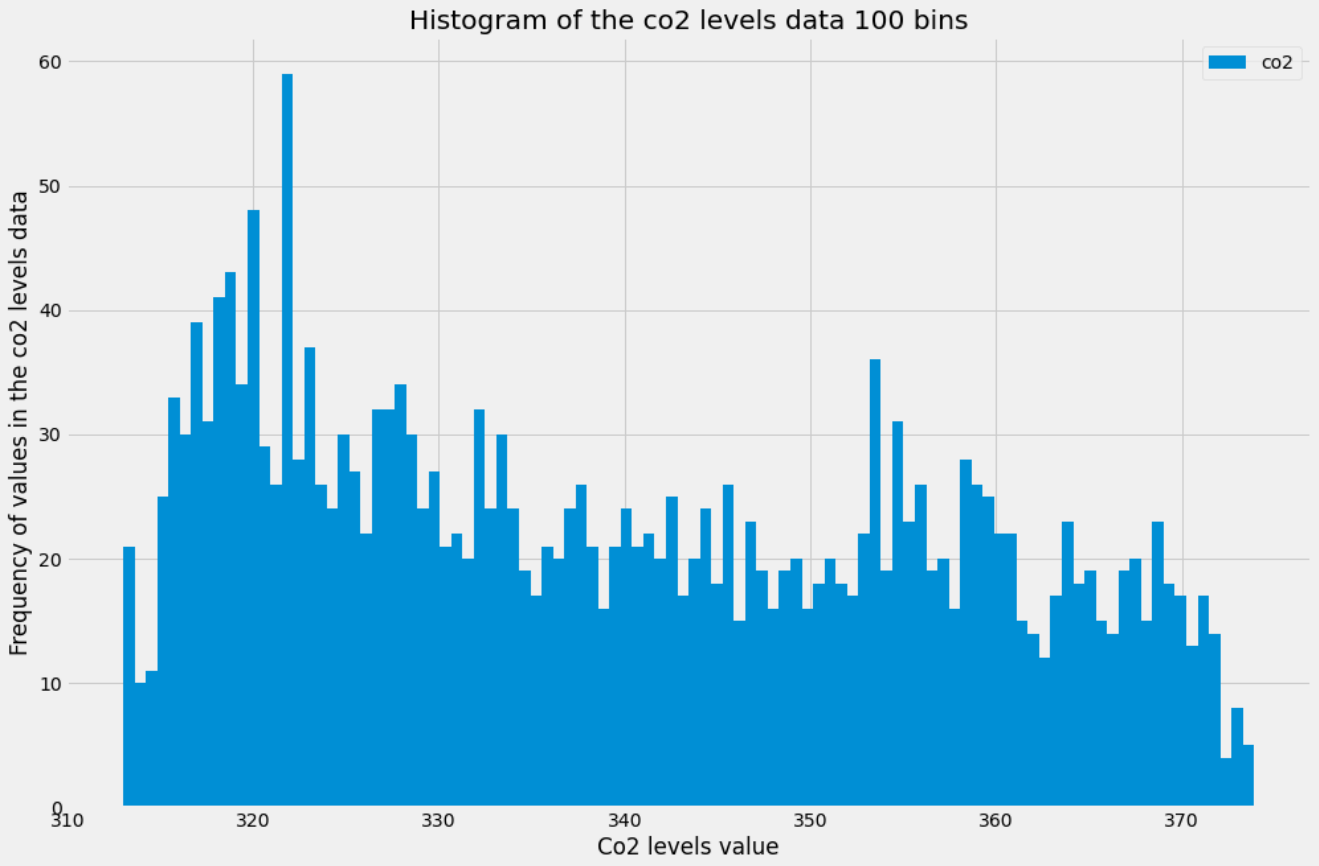


Figure 3.13. Histogram plot for the CO2 levels data.

Since it can be confusing to identify the optimal number of bins, histograms can be a cumbersome way to assess the distribution of your data. Instead, you can rely on kernel density plots to view the distribution of your data.

Kernel density plots are a variation of histograms. They use kernel smoothing to plot the values of your data, allowing for smoother distributions by dampening the effect of noise and outliers while displaying where the majority of your data is located.

It is simple to generate density plots with the pandas library, as you only need to use the standard .plot() method while specifying the kind argument as **density**.

```
# Summarizing your data with density plots
```

```
ax3 = co2_levels.plot(kind='density', linewidth=2, figsize=(15,10))
ax3.set_xlabel('co2 levels data values')
ax3.set_ylabel('Density values of the co2 levels data')
ax3.set_title('Density plot of the co2 levels data')
```

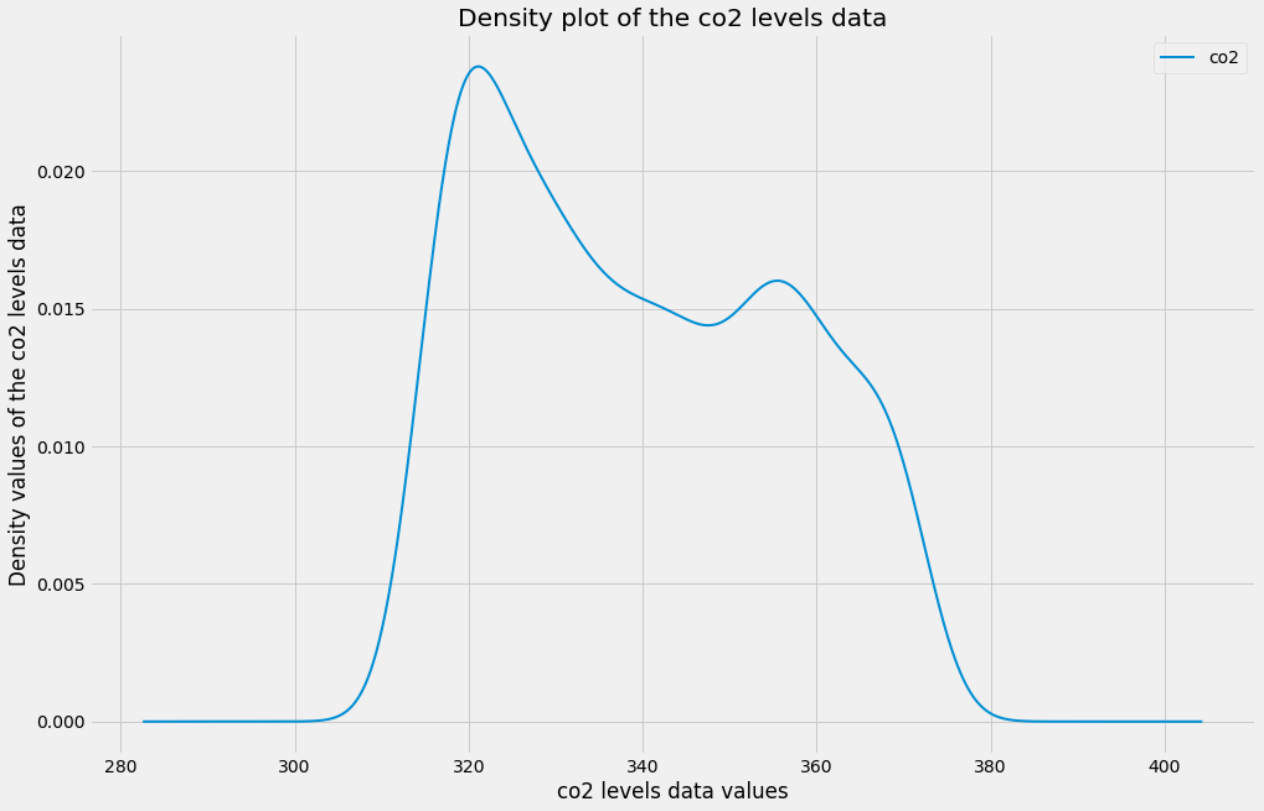


Figure 3.14.. Density plot for the CO2 levels data.

3. Visualizing Seasonality, Trend, and Noise

In this section, we will go beyond summary statistics by learning about autocorrelation and partial autocorrelation plots. You will also learn how to automatically detect seasonality, trend, and noise in your time series data. The autocorrelation and partial autocorrelation were covered in more detail in the **previous article** of this series.

3.1. Autocorrelation and Partial Autocorrelation

Autocorrelation is a measure of the correlation between your time series and a delayed copy of itself. For example, an autocorrelation of order 3 returns the correlation between a time series at points t(1), t(2), t(3), and its values lagged by 3 time points, i.e., t(4), t(5), t(6).

Autocorrelation is used to find repeating patterns or periodic signals in time series data. The principle of autocorrelation can be applied to any signal, and not just time series. Therefore, it is common to encounter the same principle in other fields, where it is also sometimes referred to as autocovariance.

In the example below, we will plot the autocorrelation of the CO2 level time series using the **plot_acf** function from the **statsmodels** library.

```
# Plotting autocorrelations
import matplotlib.pyplot as plt
from statsmodels.graphics import tsaplots
fig = tsaplots.plot_acf(co2_levels['co2'], lags=40)
plt.show()
```

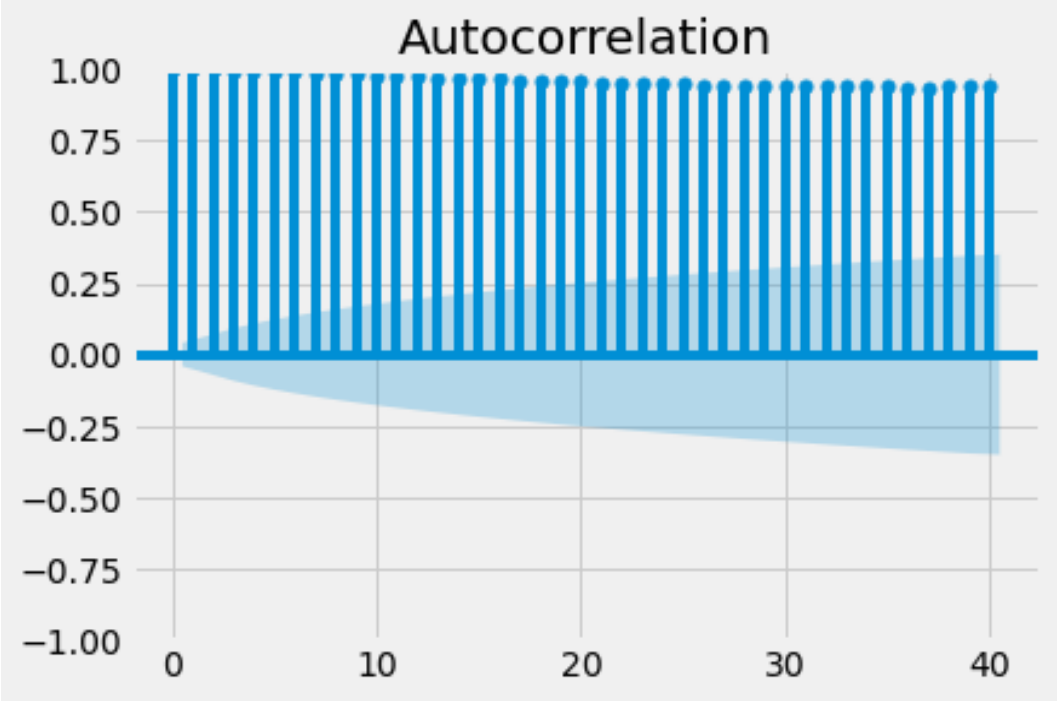


Figure 3.15. Autocorrelation for the CO2 level time series.

Since autocorrelation is a correlation measure, the autocorrelation coefficient can only take values between -1 and 1. An autocorrelation of 0 indicates no correlation, while 1 and -1 indicate strong negative and positive correlations.

In order to help you assess the significance of autocorrelation values, the **.plot_acf()** function also computes and returns margins of uncertainty, which are represented in the graph as blue-shaded regions.

Values above these regions can be interpreted as the time series having a statistically significant relationship with a lagged version of itself.
Going beyond autocorrelation, partial autocorrelation measures the correlation coefficient between a time series and lagged versions of itself. However, **it extends this idea by also removing the effect of previous time points**.

For example, a partial autocorrelation function of order 3 returns the correlation between our time series at points t(1), t(2), t(3), and lagged values of itself by 3-time points t(4), t(5), t(6), but only after removing all effects attributable to lags 1 and 2.

Just like with autocorrelation, we need to use the statsmodels library to compute and plot the partial autocorrelation in a time series. This example uses the **.plot_pacf()** function to calculate and plot the partial autocorrelation for the first 40 lags of the CO2 level time series.

```
# Plotting partial autocorrelations
import matplotlib.pyplot as plt
from statsmodels.graphics import tsaplots
fig = tsaplots.plot_pacf(co2_levels['co2'], lags=40)
plt.show()
```

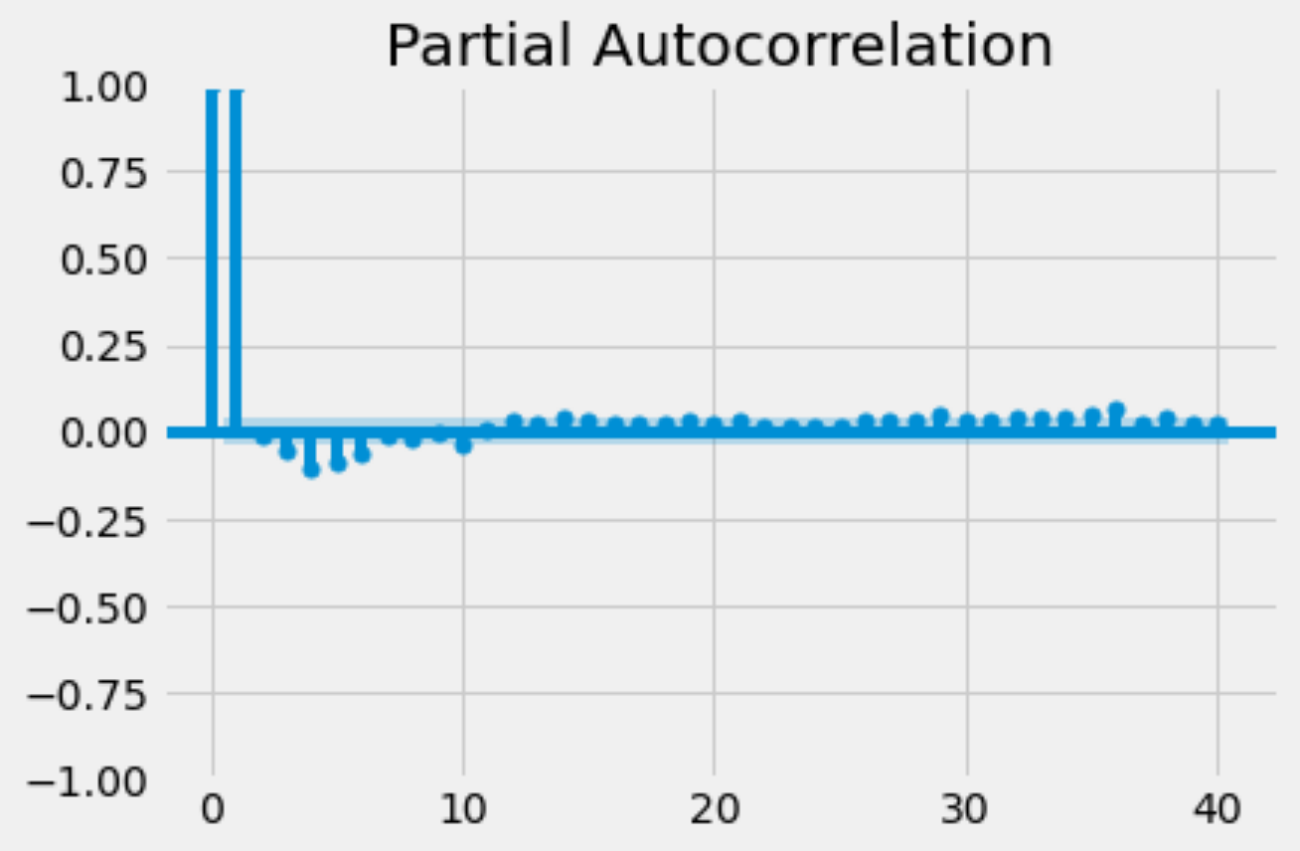


Figure 3.16. Partial autocorrelation for the CO2 level time series.

If partial autocorrelation values are close to 0, you can conclude that the values are not correlated with one another. Inversely, partial autocorrelations that have values close to 1 or -1 indicate that there exist strong positive or negative correlations between the lagged observations of the time series.
If partial autocorrelation values are beyond the margins of uncertainty, which are marked by the blue-shaded regions, then you can assume that the observed partial autocorrelation values are statistically significant.

3.2. Seasonality, trend, and noise in time series data

When looking at time-series data, you may have noticed some clear patterns that they exhibit. As you can see in the CO2 levels time series shown below, the data displays a clear upward trend as well as a periodic signal.

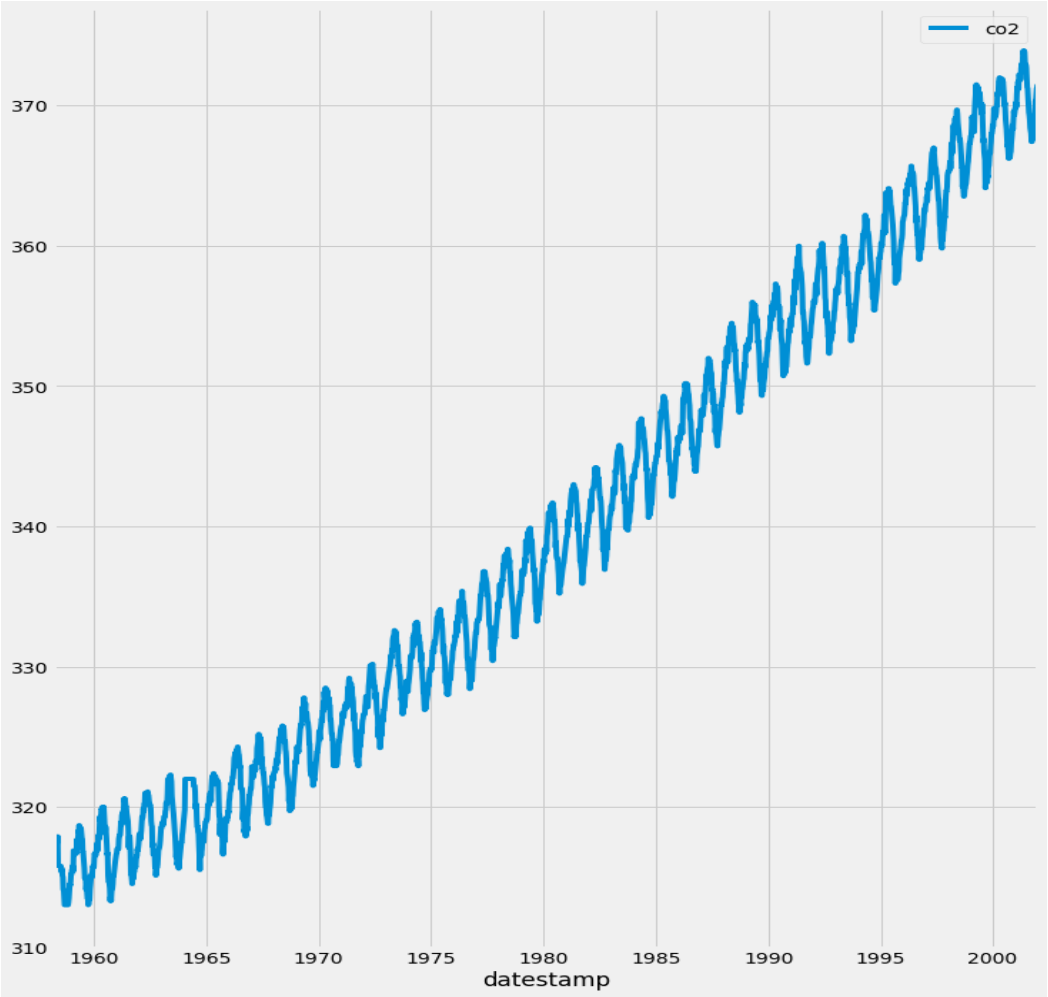


Figure 3.17. CO2 levels time series showing an upward trend.

In general, most time series can be decomposed into three major components. The first is **seasonality**, which describes the periodic signal in your time series. The second component is **trend**, which describes whether the time series is decreasing, constant, or increasing over time. Finally, the third component is **noise**, which describes the unexplained variance and volatility of your time series. Let's go through an example so that we will have a better understanding of these three components.

To decompose your time signal, we will also use the **tsa** submodule of the **statsmodels** library. The **sm.tsa.dot seasonal_decompose()** function can be used to apply time series decomposition out of the box. Let's apply it to the CO2 level data.

```
# Time series decomposition
```

```
rcParams['figure.figsize'] = 11, 15  # resizing the image to be big
enough for us
decomposition = sm.tsa.seasonal_decompose(co2_levels['co2'])
fig = decomposition.plot()
plt.show()
```

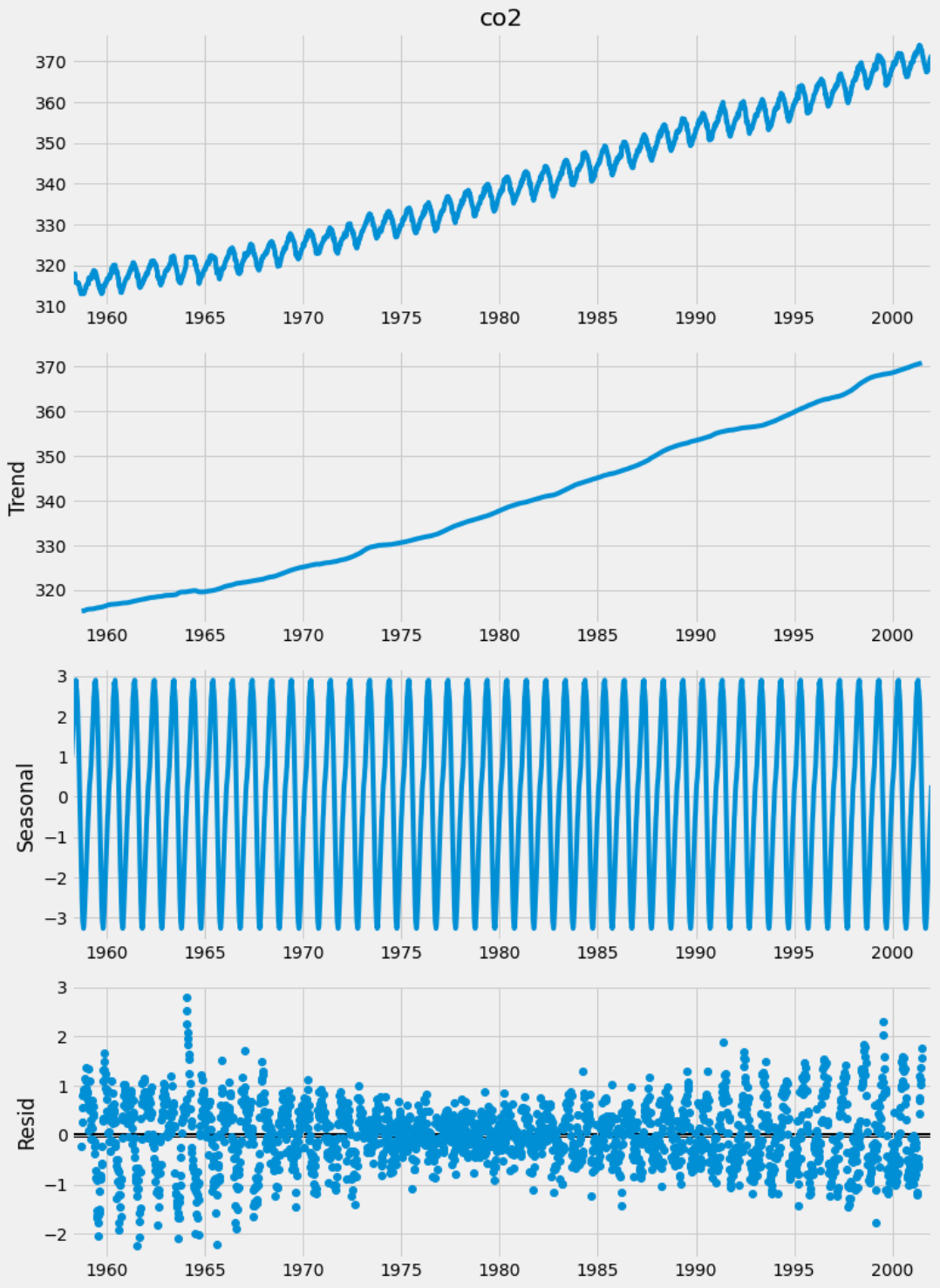


Figure 3.18. The components of the CO2 level data.

It is easy to extract each individual component and plot it. As you can see here, you can use the **dir()** command to print out the attributes associated with the decomposition variable generated before, and to print the seasonal component, use the **decomposition.seasonal** command.

```
# Seasonality component in time series

decomp_seasonal = decomposition.seasonal
ax = decomp_seasonal.plot(figsize=(14, 2))
ax.set_xlabel('Date')
ax.set_ylabel('Seasonality of time series')
ax.set_title('Seasonal values of the time series')
plt.show()
```

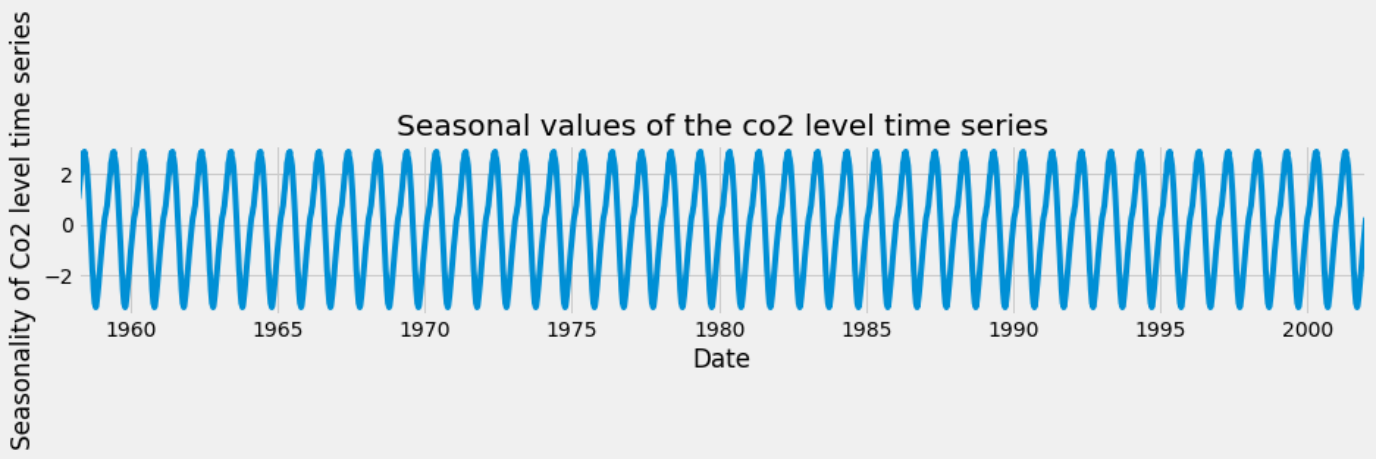


Figure 3.19. Seasonality component of the CO2 level time series.

A seasonal pattern exists when a time series is influenced by seasonal factors. Seasonality should always be a fixed and known period. For example, the temperature of the day should display clear daily seasonality, as it is always warmer during the day than at night. Alternatively, it could also display monthly seasonality, as it is always warmer in summer compared to winter.

Let's repeat the same exercise, but this time extract the trend values of the time series decomposition. The trend component reflects the overall progression of the time series and can be extracted using the decomposition **.trend()** command.

```
# Trend component in time series

decomp_trend = decomposition.trend
ax = decomp_trend.plot(figsize=(14, 2))
ax.set_xlabel('Date')
ax.set_ylabel('Trend of time series')
ax.set_title('Trend values of the time series')
plt.show()
```

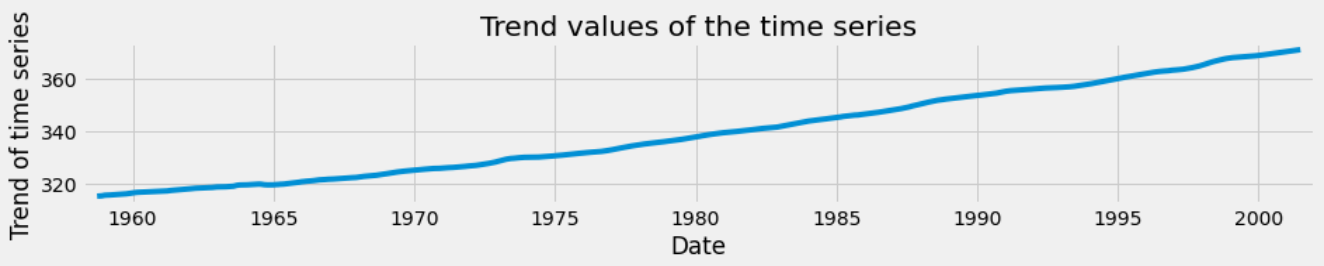


Figure 3.20. Trend component of the CO2 level time series.

Finally, you can also extract the noise, or the residual component of a time series, as shown below.

```
# Noise component in time series
decomp_resid = decomposition.resid
ax = decomp_resid.plot(figsize=(14, 2))
ax.set_xlabel('Date')
ax.set_ylabel('Residual of time series')
ax.set_title('Residual values of the time series')
plt.show()
```

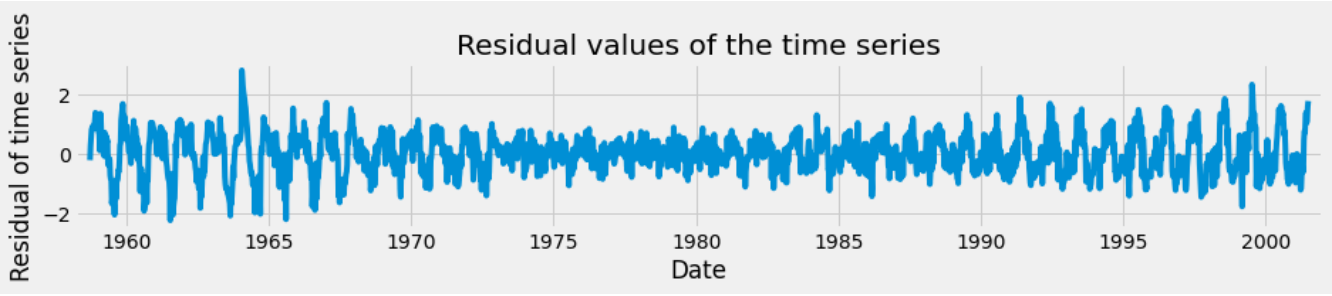


Figure 3.21. Residual component of the CO2 level time series.

The residual component describes random, irregular influences that could not be attributed to either trend or seasonality.

3.3. Analyzing airline data

You will hone your skills with the famous **airline dataset**, which consists of monthly totals of airline passengers from January 1949 to December 1960. It contains 144 data points and is often used as a standard dataset for time series analysis. **Working with this kind of data should prepare you to tackle any data that you may encounter in the real world!**

Let's first load the data and plot the number of monthly airline passengers with the code below:

```
# upload the airline data
airline = pd.read_csv('airline_passengers.csv', parse_dates=['Month'],
index_col='Month')

# Plot the time series in your DataFrame
ax = airline.plot(color='blue', fontsize=12, figsize=(12,10))
```

```
# Add a red vertical line at the date 1955-12-01
ax.axvline('1955-12-01', color='red', linestyle='--')

# Specify the labels in your plot
ax.set_xlabel('Date', fontsize=12)
ax.set_ylabel('Number of Monthly Airline Passengers', fontsize=12)
ax.set_title('Number of Monthly Airline Passengers')
plt.show()
```

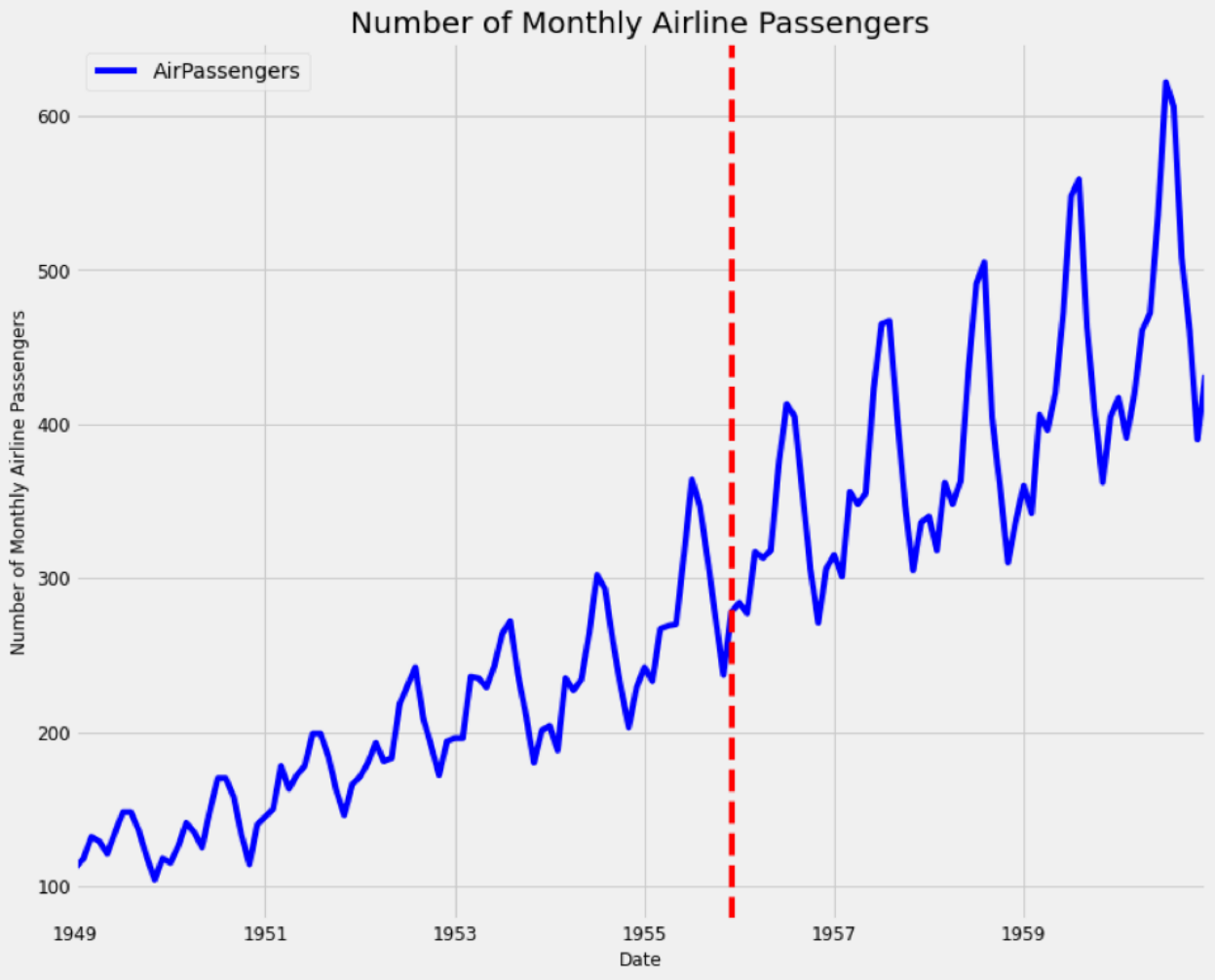


Figure 3.22. The number of monthly airline passengers.

Then we will plot the summary of the time series by printing the summary of the data and the number of missing values, and then plotting the box plot of the time series data.

```
# Print out the number of missing values
print(airline.isnull().sum())

# Print out summary statistics of the airline DataFrame
print(airline.describe())
```

```
AirPassengers    0
dtype: int64
       AirPassengers
count     144.000000
mean      280.298611
std       119.966317
min       104.000000
25%       180.000000
50%       265.500000
75%       360.500000
max       622.000000
```

```
# Display boxplot of airline values
ax = airline.boxplot()

# Specify the title of your plot
ax.set_title('Boxplot of Monthly Airline\nPassengers Count',
fontsize=20)
plt.show()
```

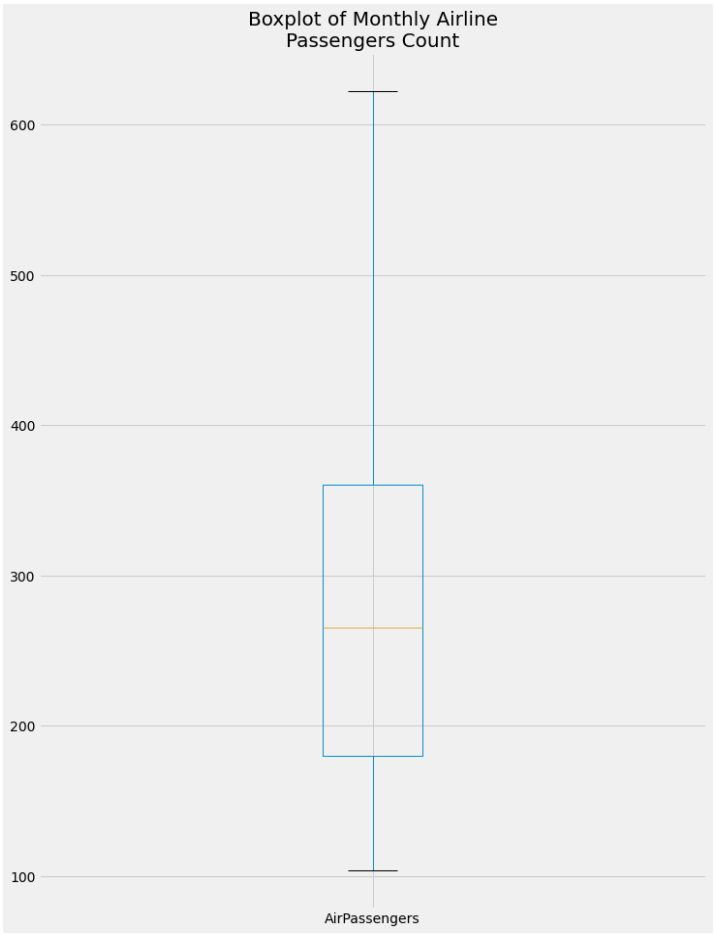


Figure 3.23. The box plot of the airline passenger time series.

From the boxplot, we can get the following information. The maximum number of monthly airline passengers exceeds 600, and the minimum is approximately 100. There are no outliers in the data. The median of the data is approximately 270, the 75th percentile is around 360, and the 25th percentile is approximately 180. Let's create and plot the monthly aggregation of the airline passengers' data.

```
# Get the month for each date from the index of the airline
index_month = airline.index.month

# Compute the mean number of passengers for each month of the year
mean_airline_by_month = airline.groupby(index_month).mean()

# Plot the mean number of passengers for each month of the year
mean_airline_by_month.plot()
plt.legend(fontsize=20)
plt.show()
```

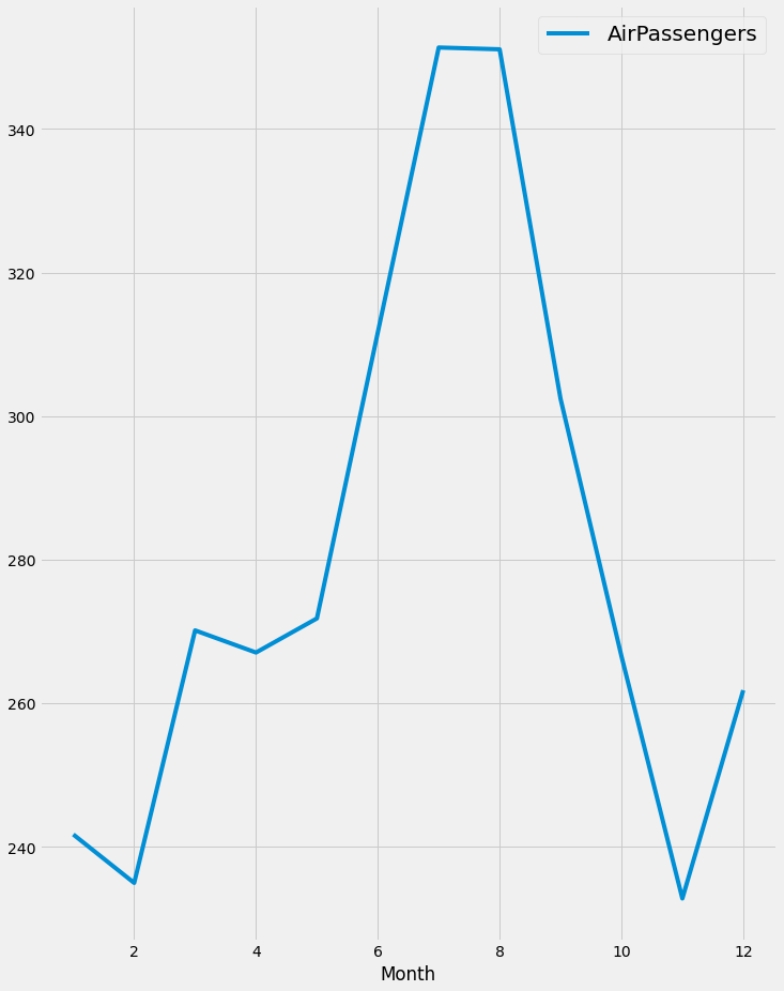


Figure 3.24. Monthly aggregation of the airline passengers' data.

There is a rise in the number of airline passengers in July and August, which is reasonable due to the vacation period during this time. Finally, we will decompose the time series and plot the trend and seasonality in the data.

```
# Import statsmodels.api as sm
import statsmodels.api as sm

# Perform time series decompositon
decomposition = sm.tsa.seasonal_decompose(airline)

# Extract the trend and seasonal components
trend = decomposition.trend
seasonal = decomposition.seasonal

airline_decomposed = pd.DataFrame({'Trend':trend,
'Seasonal':seasonal})

# Print the first 5 rows of airline_decomposed
print(airline_decomposed.head(5))

# Plot the values of the airline_decomposed DataFrame
ax = airline_decomposed.plot(figsize=(12, 6), fontsize=15)

# Specify axis labels
ax.set_xlabel('Date', fontsize=15)
plt.legend(fontsize=15)
plt.show()
```

The trend in Figure 3.25 shows that the number of passengers is increasing over the years 1949 to 1959, which is reasonable as the number of airplanes themselves increased. There is also seasonality in the data, which is expected, as shown in the monthly aggregation plot.

4. Visualizing Multiple Time Series

In the field of Data Science, it is common to be involved in projects where multiple time series need to be studied simultaneously. In this section, we will show you how to plot multiple time series at once and how to discover and describe relationships between multiple time series.
In this section, we will be working with a new dataset that contains volumes of different types of meat produced in the United States between 1944 and 2012. The dataset can be downloaded from **here**.

4.1 Working with more than one time series

In the field of data science, you will often come across datasets containing multiple time series. For example, we could be measuring the performance of CPU servers over time, and in another case, we could be exploring the stock performance of different companies over time.
These situations introduce several different questions and therefore require additional analytical tools and visualization techniques.

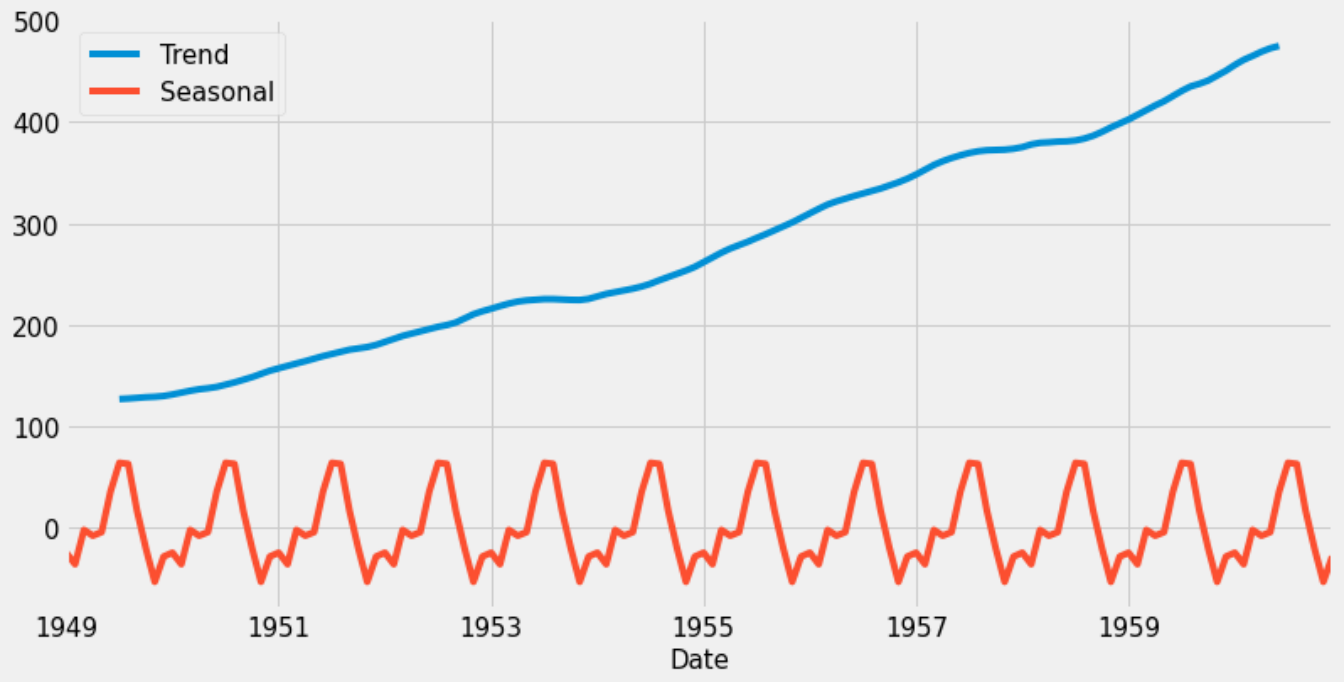


Figure 3.25. The trend and seasonality in the time series of passengers.

A convenient aspect of pandas is that dealing with multiple time series is very similar to dealing with a single time series. Just like in the previous sections, you can quickly leverage the **.plot()** and **.describe()** methods to visualize and produce statistical summaries of the data.

```
meat = pd.read_csv("meat.csv", parse_dates=['date'], index_col='date')
print(meat.head(5))
```

```
AirPassengers    0
dtype: int64
       AirPassengers
count     144.000000
mean      280.298611
std       119.966317
min       104.000000
25%       180.000000
50%       265.500000
75%       360.500000
max       622.000000
```

```
# plotting multiple time series
ax = meat.plot(figsize=(15, 10), fontsize=14)
plt.show()
```

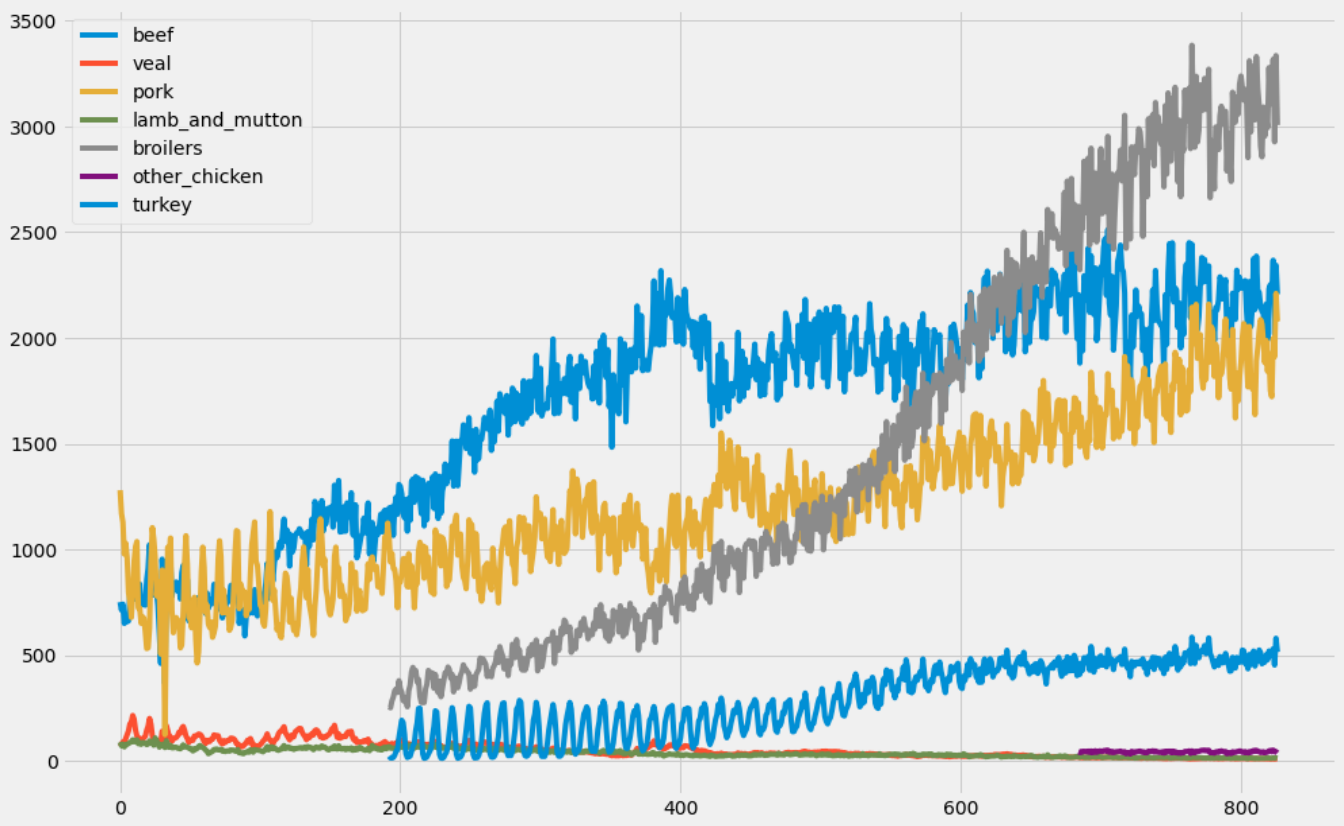


Figure 3.26. Line plots for the volumes of different types of meat were produced in the United States between 1944 and 2012.

Another interesting way to plot multiple time series is to use area charts. Area charts are commonly used when dealing with multiple time series and can be leveraged to represent cumulated totals. With the pandas library, you can simply leverage the **.area()** method as to produce an area chart as shown in Figure 3.27.

```
# Area charts
ax = meat.plot.area(figsize=(15, 10), fontsize=14)
```

4.2. Plot multiple time series

When plotting multiple time series, matplotlib will iterate through its default color scheme until all columns in the DataFrame have been plotted. Therefore, the repetition of the default colors may make it difficult to distinguish some of the time series.

For example, since there are seven time series in the meat dataset, some time series are assigned the same blue color. In addition, matplotlib does not consider the color of the background, which can also be an issue.

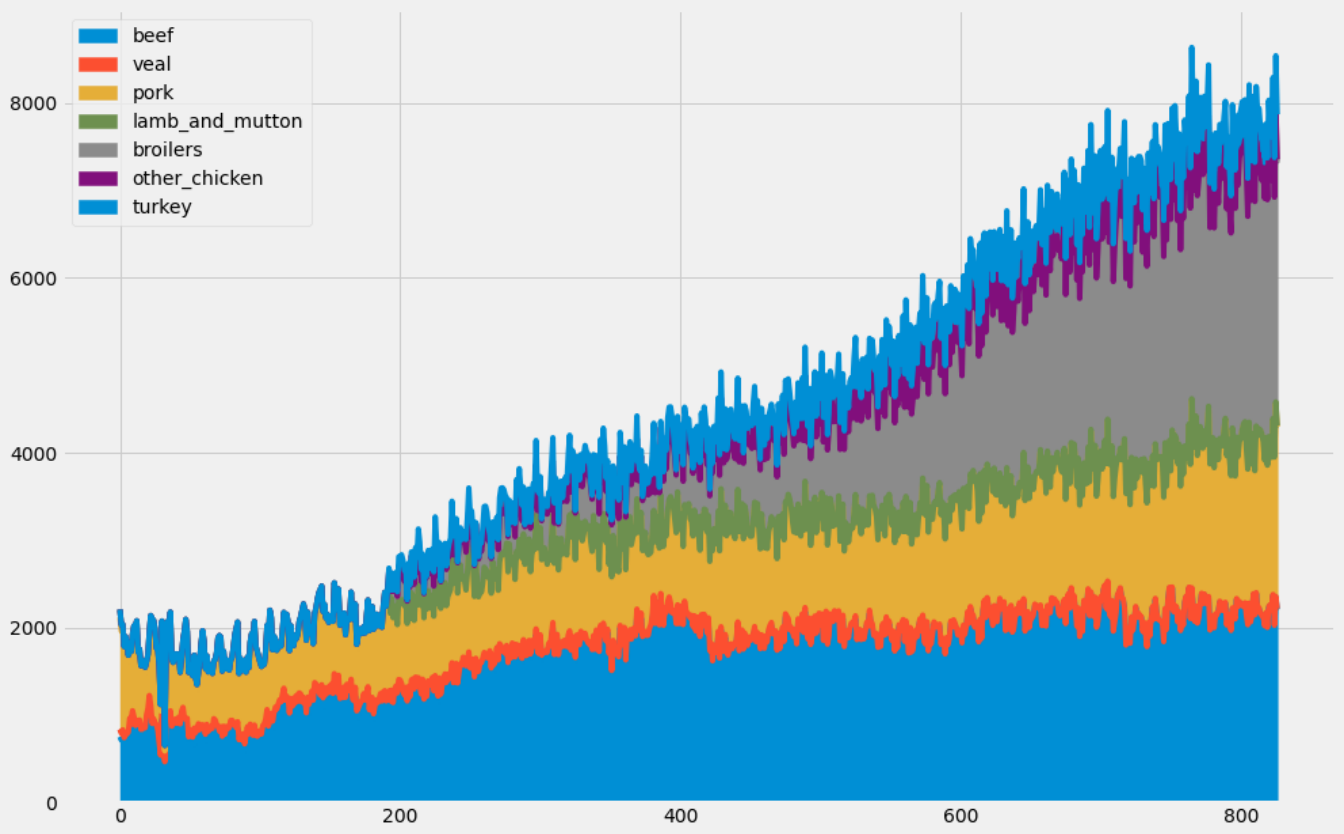


Figure 3.27. Area plot of the volumes of different types of meats was produced in the United States between 1944 and 2012.

To remedy this, the **.plot()** method has an additional argument called colormap. This argument allows you to assign a wide range of color palettes with varying contrasts and intensities. You can either define your own Matplotlib colormap or use a string that matches a colormap registered with matplotlib. In this example, we use the **Dark2** color palette and the result is shown in Figure 3.28.

```
ax = meat.plot(colormap='Dark2', figsize=(14, 7))
ax.set_xlabel('Date')
ax.set_ylabel('Production Volume (in tons)')
plt.show()
```

When building slides for a presentation or sharing plots with stakeholders, it can be more convenient for yourself and others to visualize both time series plots and numerical summaries on a single graph. In order to do so, first plot the columns of your DataFrame and return the matplotlib AxesSubplot object to the variable ax.

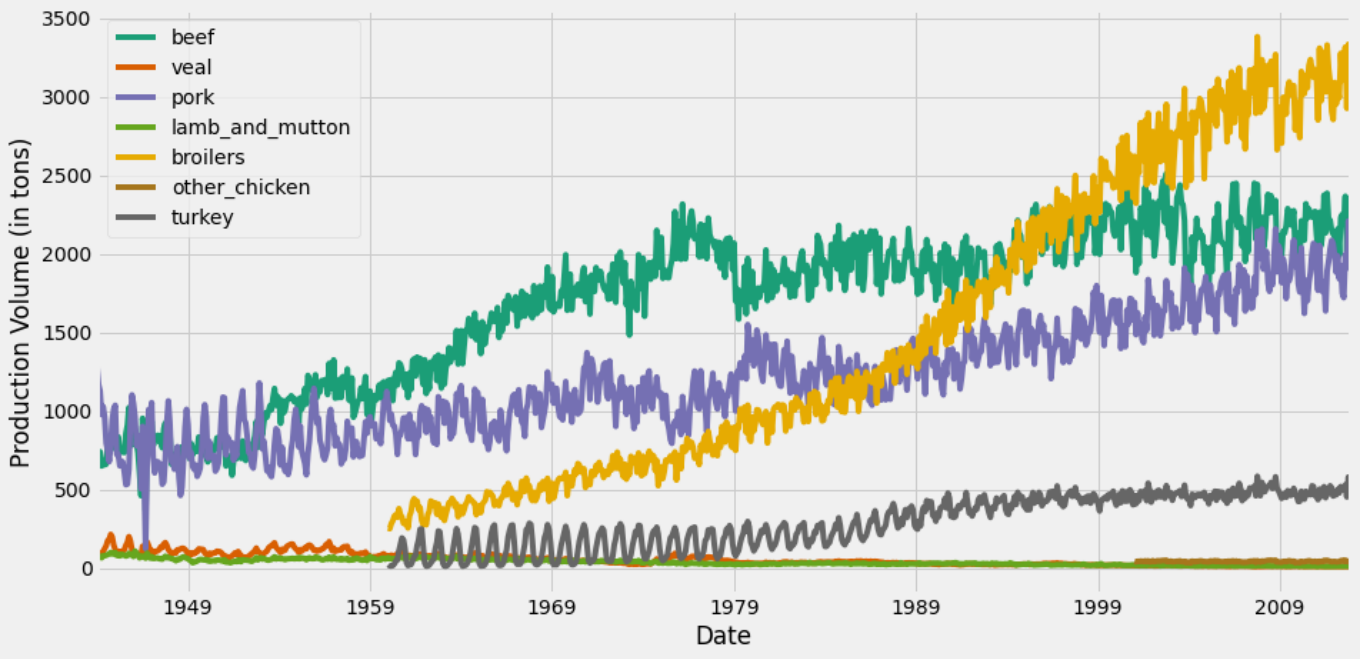
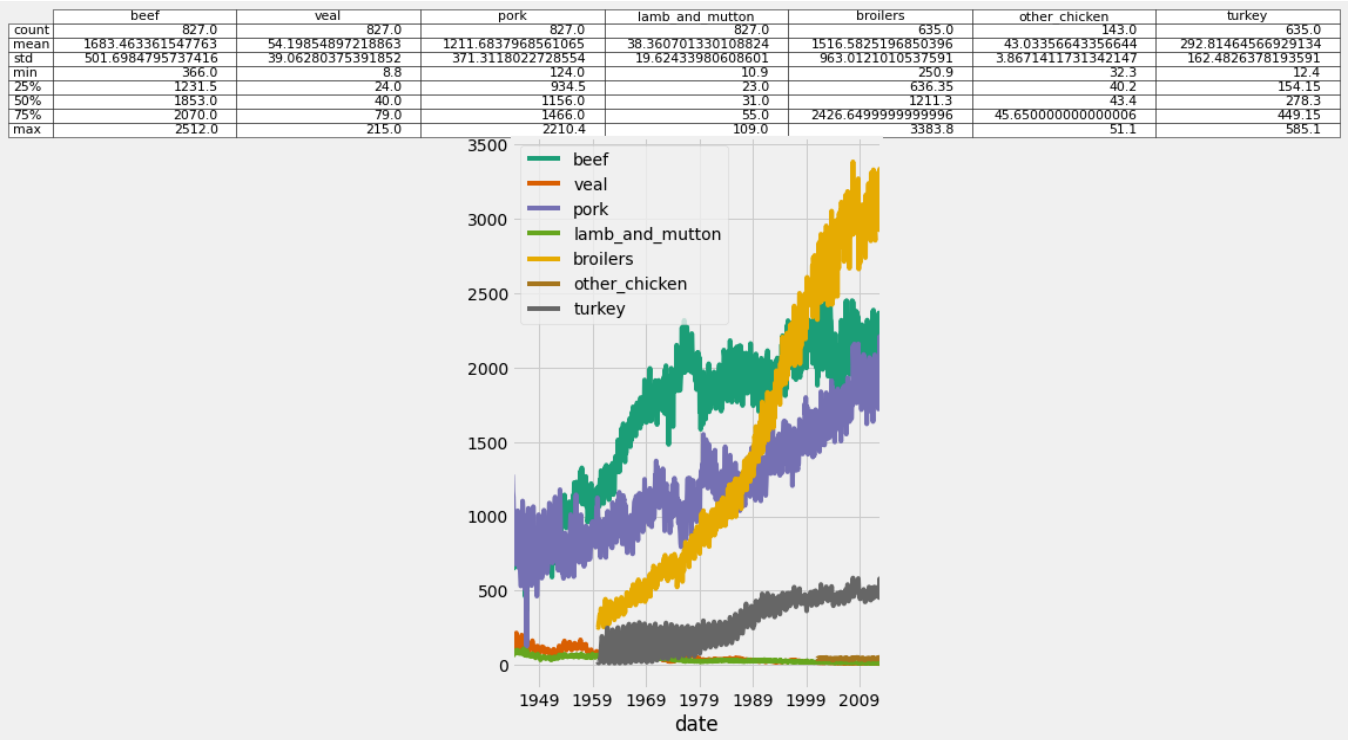


Figure 3.28. Line plots for the volumes of different types of meats were produced in the United States between 1944 and 2012 with the Dark2 color palette.

You can then pass any table information in pandas as a DataFrame or Series to the ax object. Here we obtain summary statistics of the DataFrame by using the **.describe()** method and then pass this content as a table with the ax. dot table command as shown in Figure 3.29.



	beef	veal	pork	lamb_and_mutton	broilers	other_chicken	turkey
count	827.0	827.0	827.0	827.0	635.0	143.0	635.0
mean	1683.463361547763	54.19854897218863	1211.6837968561065	38.360701330108824	1516.5825196850396	43.03356643356644	292.81464566929134
std	501.6984795737416	39.06280375391852	371.3118022728554	19.62433980608601	963.0121010537591	3.8671411731342147	162.4826378193591
min	366.0	8.8	124.0	10.9	250.9	32.3	12.4
25%	1231.5	24.0	934.5	23.0	636.35	40.2	154.15
50%	1853.0	40.0	1156.0	31.0	1211.3	43.4	278.3
75%	2070.0	79.0	1466.0	55.0	2426.6499999999996	45.650000000000006	449.15
max	2512.0	215.0	2210.4	109.0	3383.8	51.1	585.1

Figure 3.29. Summary statistics of the and line plots of meats produced in the United States between 1944 and 2012.

In order to overcome issues with visualizing datasets containing time series of different scales, you can leverage the subplots argument, which will plot each column of a DataFrame on a different subplot. In addition, the layout of your subplots can be specified using the layout keyword, which accepts two integers specifying the number of rows and columns to use.

It is important to ensure that the total number of subplots is greater than or equal to the number of time series in your DataFrame. You can also specify if each subgraph should share the values of its x-axis and y-axis using the **sharex** and **sharey** arguments. Finally, you need to specify the total size of your graph (which will contain all subgraphs) using the **figsize** argument.

```
# Facet plots
meat.plot(subplots=True, linewidth=0.5, layout=(2, 4), figsize=(16,
12), sharex=False, sharey=False)
plt.show()
```

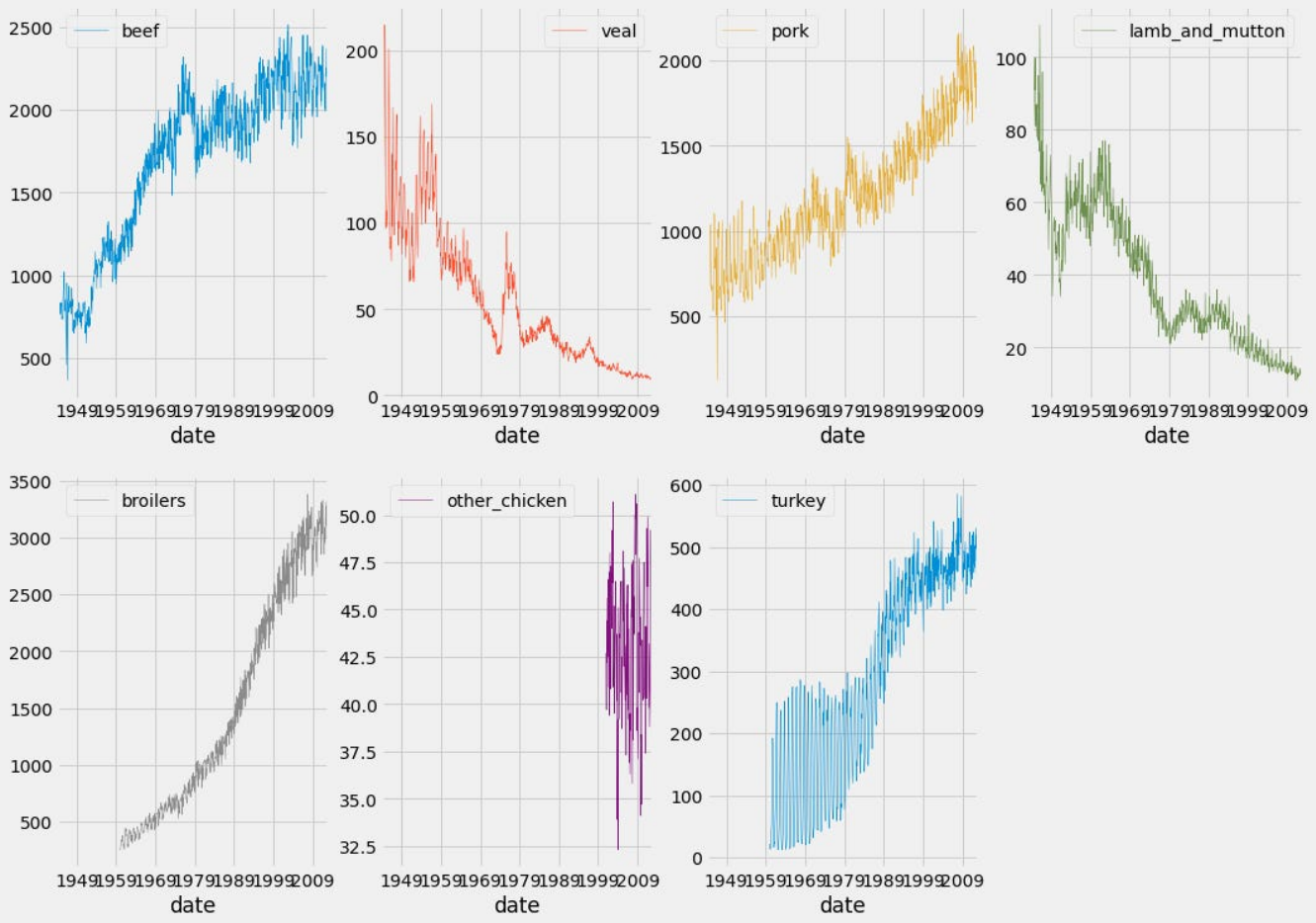


Figure 3.30. Facet plots of the meats produced in the United States between 1944 and 2012

4.3. Visualizing the relationships between multiple time series

One of the most widely used methods to assess the similarities between a group of time series is by using the correlation coefficient. The correlation coefficient is a measure used to determine the strength or lack of relationship between two variables.

The standard way to compute correlation coefficients is by using Pearson's coefficient, **which should be used when you think that the relationship between your variables of interest is linear.** Otherwise, you can **use the Kendall Tau or Spearman rank coefficient methods when the relationship between your variables of interest is thought to be non-linear.**

In Python, you can quickly compute the correlation coefficient between two variables by using the **pearsonr**, **spearmanr,** or **kendalltau** functions in the **scipy.stats.stats** module. All three of these correlation measures return both the correlation and p-value between the two variables x and y.

If you want to investigate the dependence between multiple variables at the same time, you will need to compute a correlation matrix. The result is a table containing the correlation coefficients between each pair of variables. Correlation coefficients can take any values between -1 and 1. A correlation of 0 indicates no correlation, while 1 and -1 indicate strong positive and negative correlations.

The pandas library comes in with a **.corr()** method that allows you to measure the correlation between all pairs of columns in a DataFrame. Using the meat dataset, we selected the columns beef, veal, and turkey and invoked the **.corr()** method by invoking both the Pearson and Spearman methods. The results are correlation matrices stored as two new pandas DataFrames called corr_p and corr_s.

```
# Computing Correlation Matrices
from scipy.stats.stats import pearsonr
from scipy.stats.stats import spearmanr
from scipy.stats.stats import kendalltau

corr_p = meat[['beef', 'veal','turkey']].corr(method='pearson')
print('Pearson correlation matrix')
print(corr_p)

corr_s = meat[['beef', 'veal','turkey']].corr(method='spearman')
print('Spearman correlation matrix')
print(corr_s)
```

```
Pearson correlation matrix
             beef      veal    turkey
beef     1.000000 -0.829704  0.738070
veal    -0.829704  1.000000 -0.768366
turkey   0.738070 -0.768366  1.000000
Spearman correlation matrix
             beef      veal    turkey
beef     1.000000 -0.812437  0.778533
veal    -0.812437  1.000000 -0.829492
turkey   0.778533 -0.829492  1.000000
```

Once you have stored your correlation matrix in a new DataFrame, it might be easier to visualize it instead of trying to interpret several correlation coefficients at once. In order to achieve this, we will introduce the Seaborn library, which will be used to produce a heatmap of our correlation matrix as shown in Figure 3.31.

```
import seaborn as sns
corr_mat = meat.corr(method='pearson')
sns.heatmap(corr_mat)
```

Heatmap is a useful tool to visualize correlation matrices, but the lack of order can make it difficult to read or even identify which groups of time series are the most similar.

For this reason, it is recommended to leverage the **.clustermap()** function in the seaborn library, which applies hierarchical clustering to your correlation matrix to plot a sorted heatmap, where similar time series are placed closer to one another, as shown in Figure 3.32.

```
# Clustermap
sns.clustermap(corr_mat)
```

5. Case Study: Unemployment Rate

In this section, we will practice all the concepts covered in the course. We will visualize the **unemployment rate** in the US from 2000 to 2010. The jobs dataset contains time series for 16 industries across a total of 122 time points, one per month for 10 years.

5.1. Explore the data

The first step in data exploration is to print the summary statistics and plot the summary of the data using a boxplot as shown in Figure 3.33.

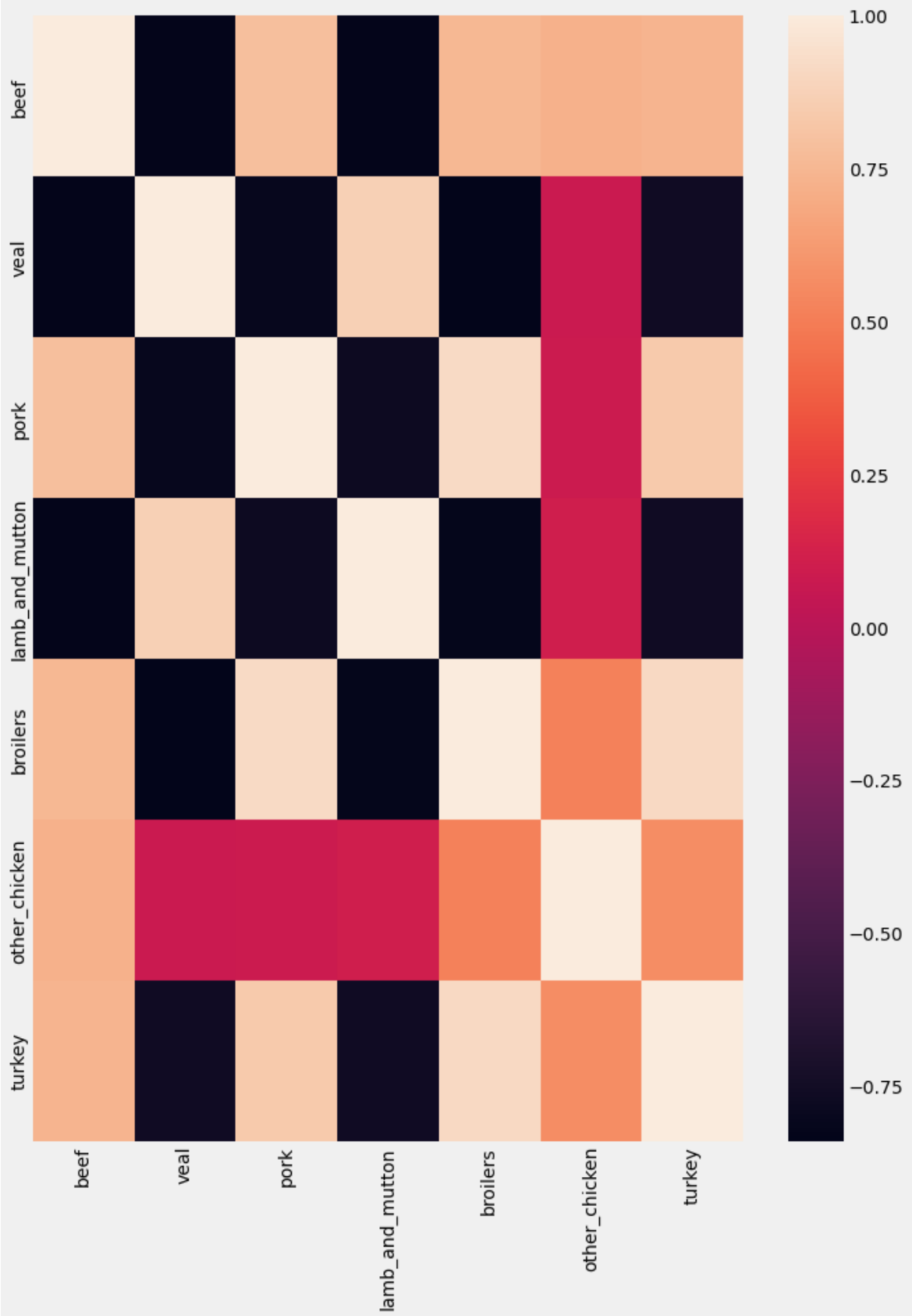


Figure 3.31. The heatmap for the correlation matrix of the meat dataset.

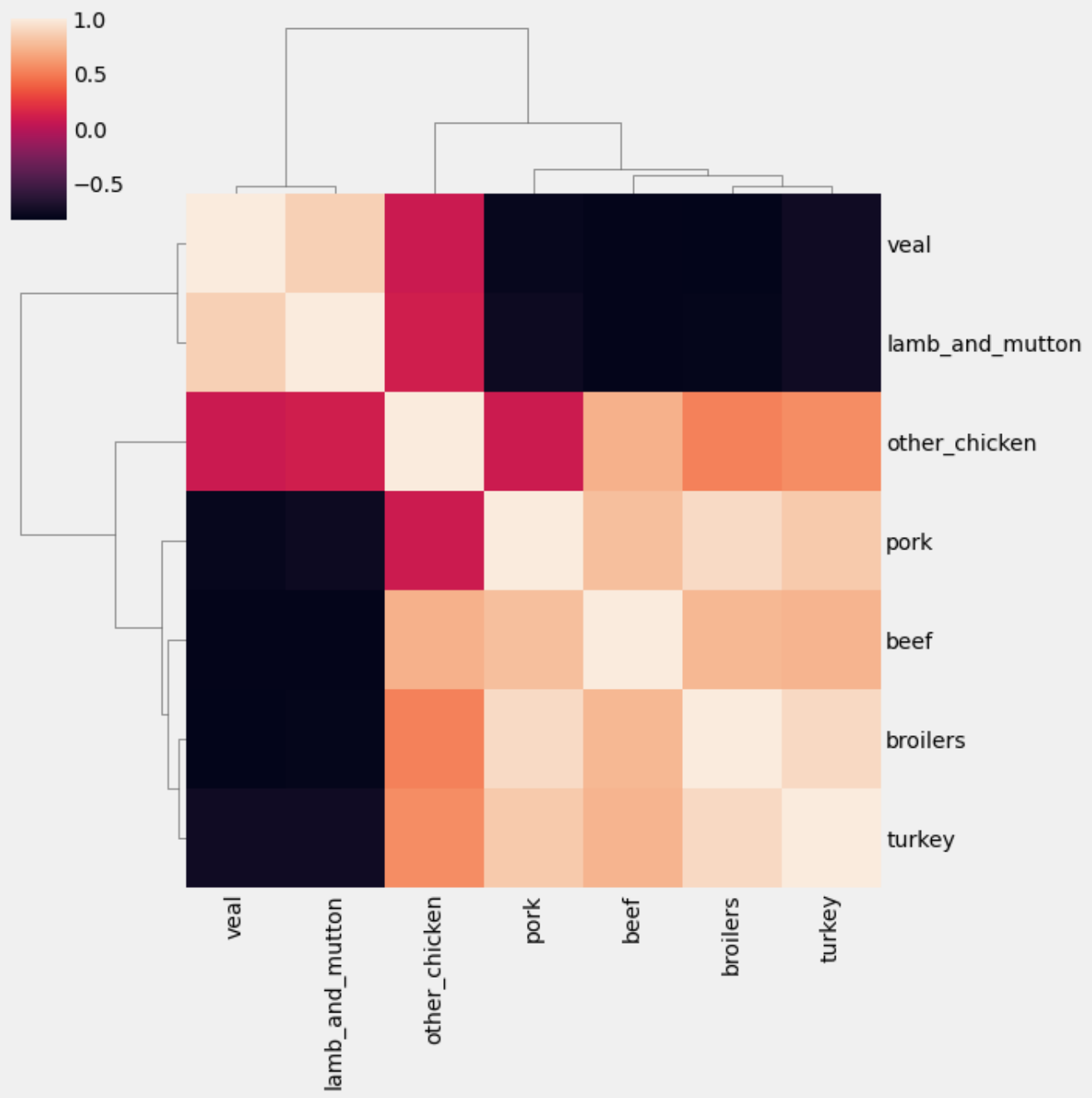


Figure 3.32. Hierarchical clustering of the meat dataset correlation.

```
# Read in jobs file
jobs = pd.read_csv('employment.csv')

# Print first five lines of your DataFrame
print(jobs.head(5))

# Check the type of each column in your DataFrame
print(jobs.dtypes)
```

```
# Convert datestamp column to a datetime object
jobs['datestamp'] = pd.to_datetime(jobs['datestamp'])

# Set the datestamp columns as the index of your DataFrame
jobs = jobs.set_index('datestamp')

# Check the number of missing values in each column
print(jobs.isnull().sum())

# Generate a boxplot
jobs.boxplot(fontsize=6, vert=False)
plt.show()

# Generate numerical summaries
print(jobs.describe())

# Print the name of the time series with the highest mean
print('Agriculture')

# Print the name of the time series with the highest variability
print('Construction')
```

We can also plot a line plot for each feature in one facet plot as follows and the result is shown in Figure 3.34.

```
# Facet plots of the jobs dataset

jobs.plot(subplots=True, layout=(4, 4), figsize=(30, 16), sharex=True,
sharey=False)
plt.show()
```

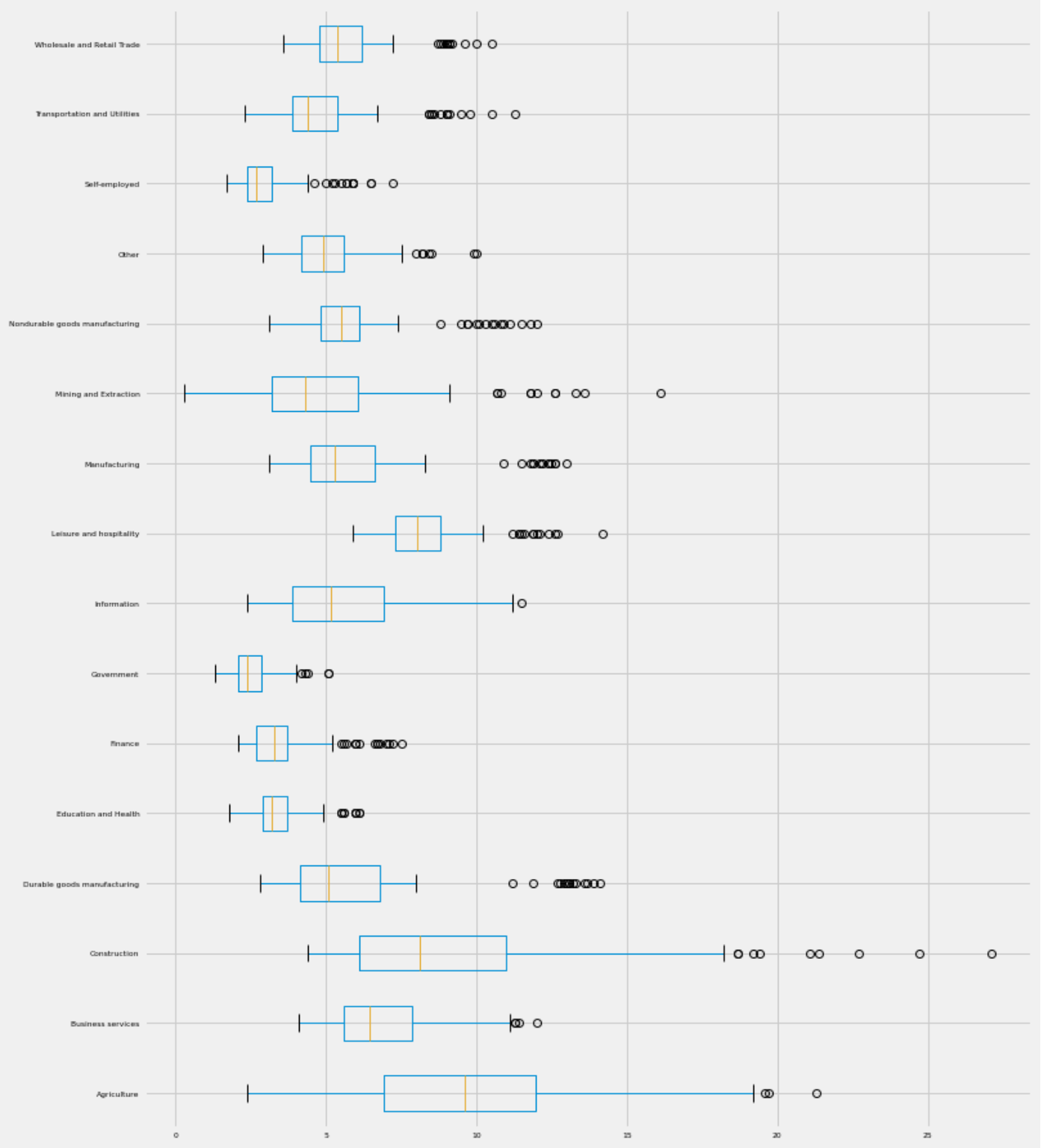


Figure 3.33. Box plot for the unemployment data.

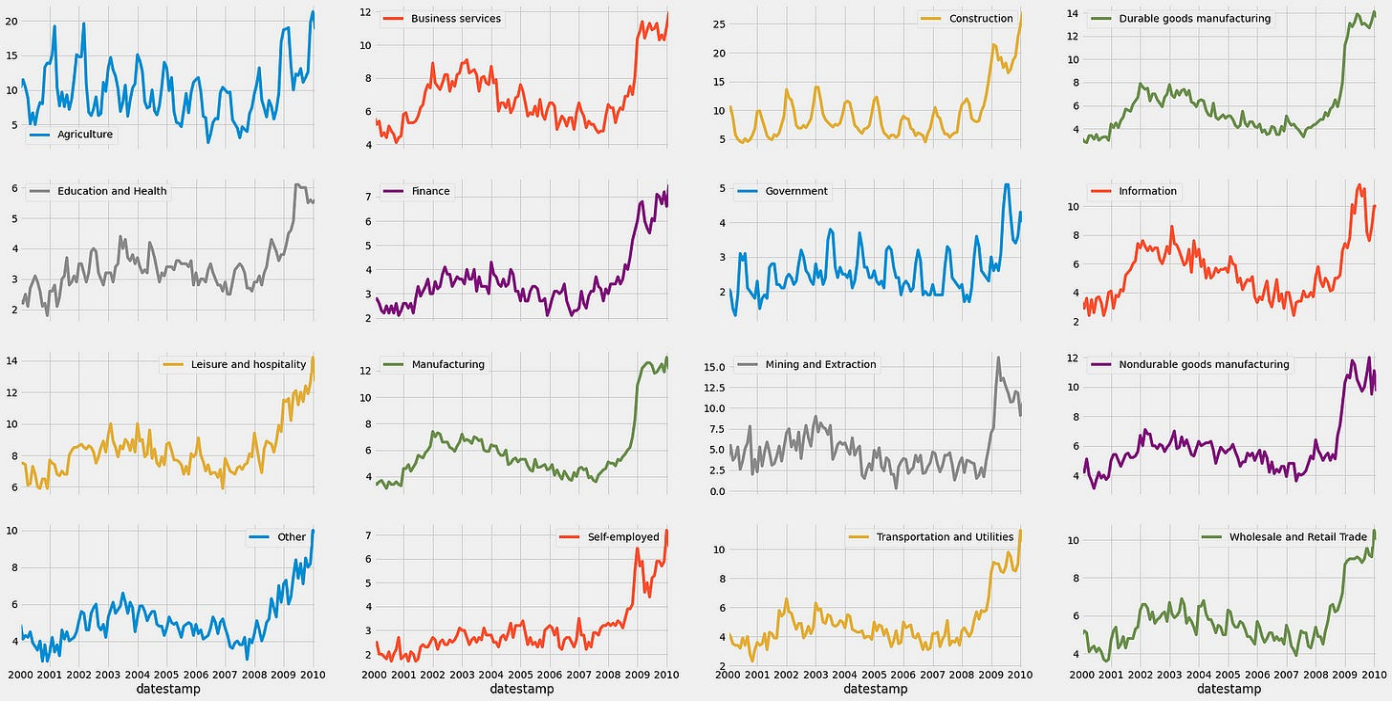


Figure 3.34. Facet plots of the unemployment dataset.

As you can see, the unemployment rate in the USA skyrocketed after the 2008 financial crisis. It is impressive to see how all industries were affected! Since 2008 appears to be the year when the unemployment rate in the USA started increasing, let's annotate our plot with vertical lines using the familiar **axvline** notation shown in Figure 3.35.

```
# Annotating events in the jobs dataset

ax = jobs.plot(figsize=(20, 14), colormap='Dark2')
ax.axvline('2008-01-01', color='black', linestyle='--')
ax.axvline('2009-01-01', color='black', linestyle='--')
```

We can also calculate and plot the monthly or daily average of the unemployment rate for each job section and plot it as shown in Figure 3.36.

```
# Monthly averages in the jobs dataset

index_month = jobs.index.month
jobs_by_month = jobs.groupby(index_month).mean()
print(jobs_by_month)
ax = jobs_by_month.plot(figsize=(12, 5), colormap='Dark2')
ax.legend(bbox_to_anchor=(1.0, 0.5), loc='center left')
```

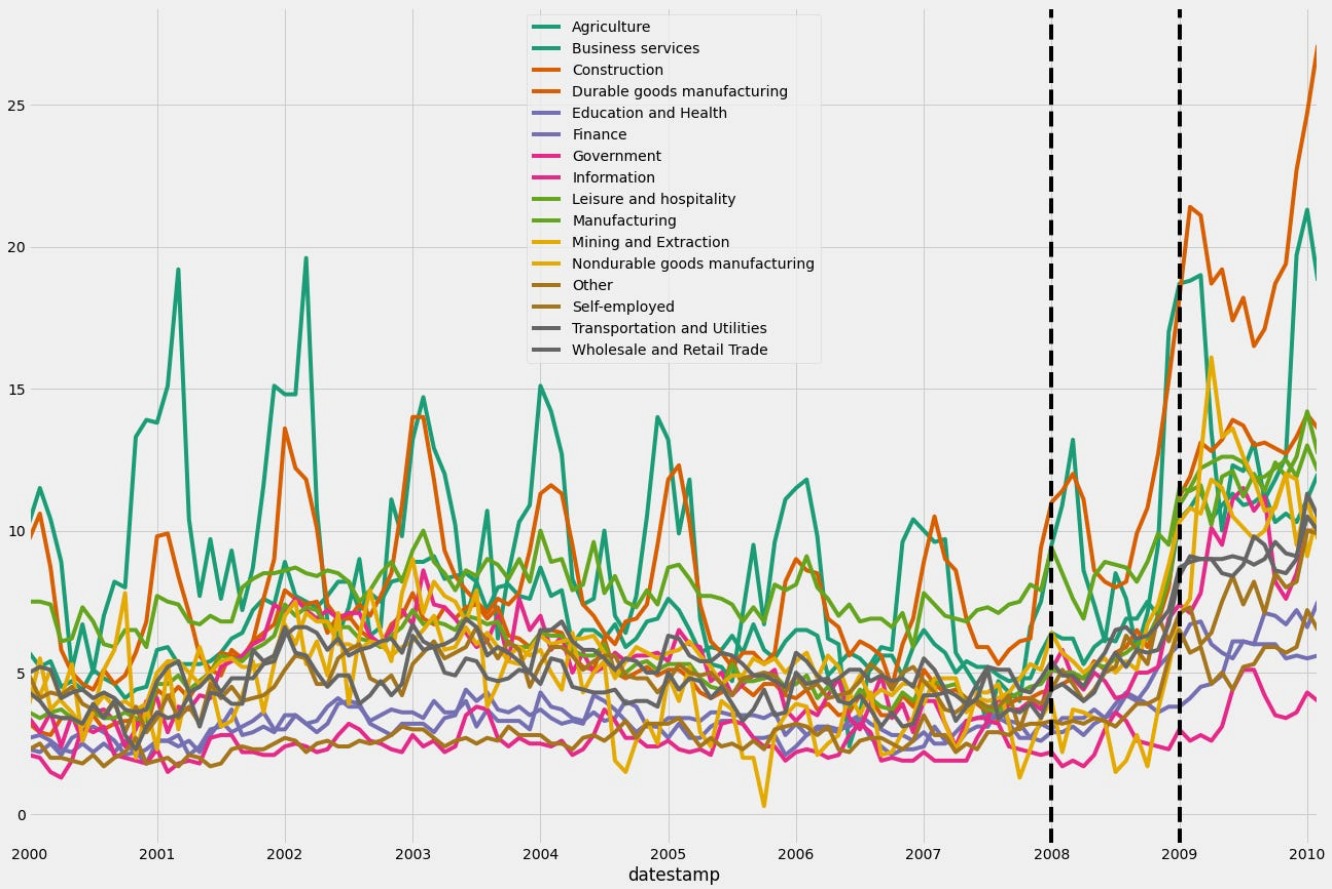


Figure 35. Line plot for the unemployment data over time.

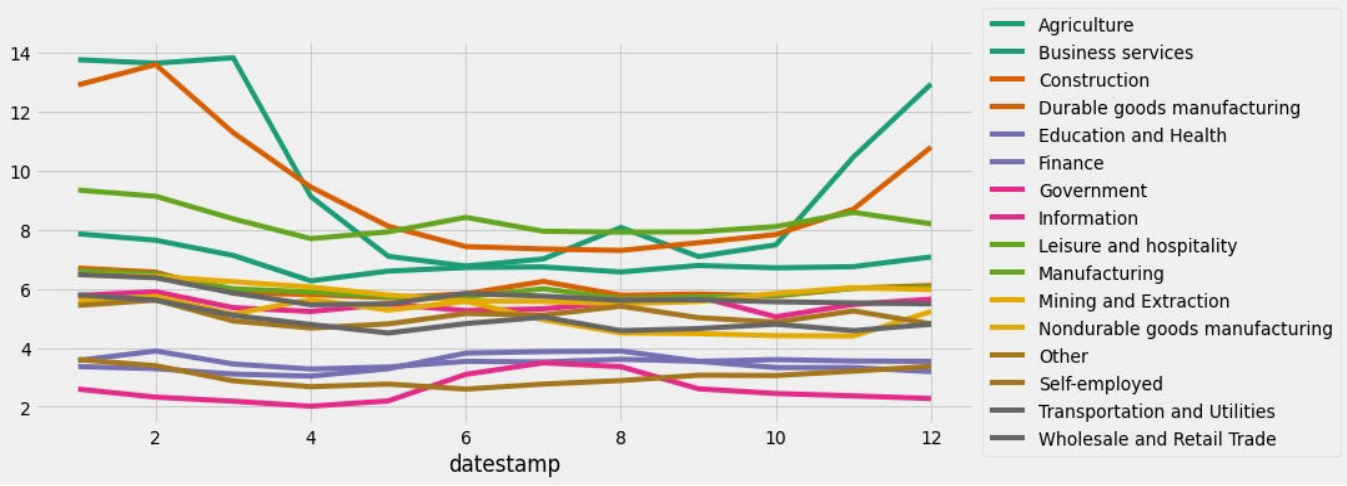


Figure 36. The monthly average for the unemployment rate in each job sector.

The resulting plot shows some interesting patterns! For example, the unemployment rate for the Agriculture and Construction industries shows significant peaks during the winter months, which

is consistent with the idea that these industries will be far less active during the cold weather months!

5.2. Seasonality, trend, and noise in the time-series data

In the previous subsection, we extracted interesting patterns and seasonality from some of the time series in the jobs dataset. In section 3, the concept of time series decomposition was introduced, which allows us to automatically extract the seasonality, trend, and noise of the time series.

In the code below, we will begin by initializing a my_dict dictionary and extracting the column names of the jobs dataset.

```
# Decomposing multiple time series with Python dictionaries
# Import the statsmodel library

import statsmodels.api as sm
# Initialize a dictionary
my_dict = {}
# Extract the names of the time series
ts_names = jobs.columns
print(ts_names)
```

Then, we will use a “for” loop to iterate through the columns of df and apply the seasonal_decompose() function from the statsmodels library, which is stored in my_dict. Then we will extract the trend component and store it in a new Dataframe and plot it.

```
# Run time series decomposition
for ts in ts_names:
    ts_decomposition = sm.tsa.seasonal_decompose(jobs[ts])
    my_dict[ts] = ts_decomposition

# Initialize a new dictionary for each component
my_dict_trend = {}
my_dict_seasonal = {}
my_dict_resid = {}

# Extract the trend component
for ts in ts_names:
    my_dict_trend[ts] = my_dict[ts].trend
    my_dict_seasonal[ts] = my_dict[ts].seasonal
    my_dict_resid[ts] = my_dict[ts].resid

# Convert to a DataFrame
trend_df = pd.DataFrame.from_dict(my_dict_trend)
seasonal_df = pd.DataFrame.from_dict(my_dict_seasonal)
resid_df = pd.DataFrame.from_dict(my_dict_resid)
```

Let's now plot the three components. First is the seasonal component of the jobs dataset:

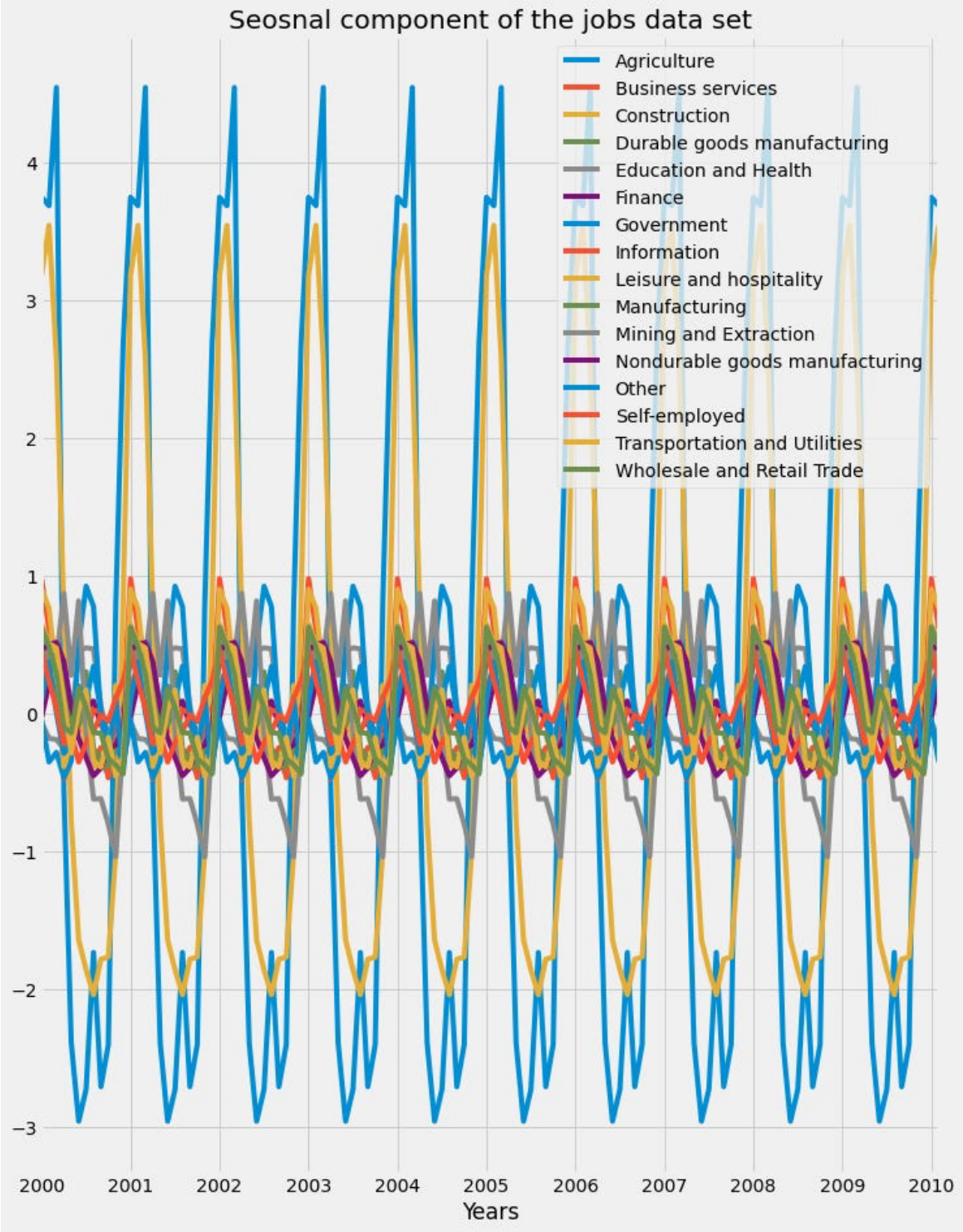


Figure 3.37. Seasonal component of the job's dataset.

We can see that certain industries were more affected by seasonality than others, as we saw that the Agriculture and Construction industries saw rises in unemployment rates during the colder months of winter. Next, the trend component of the jobs dataset is plotted:

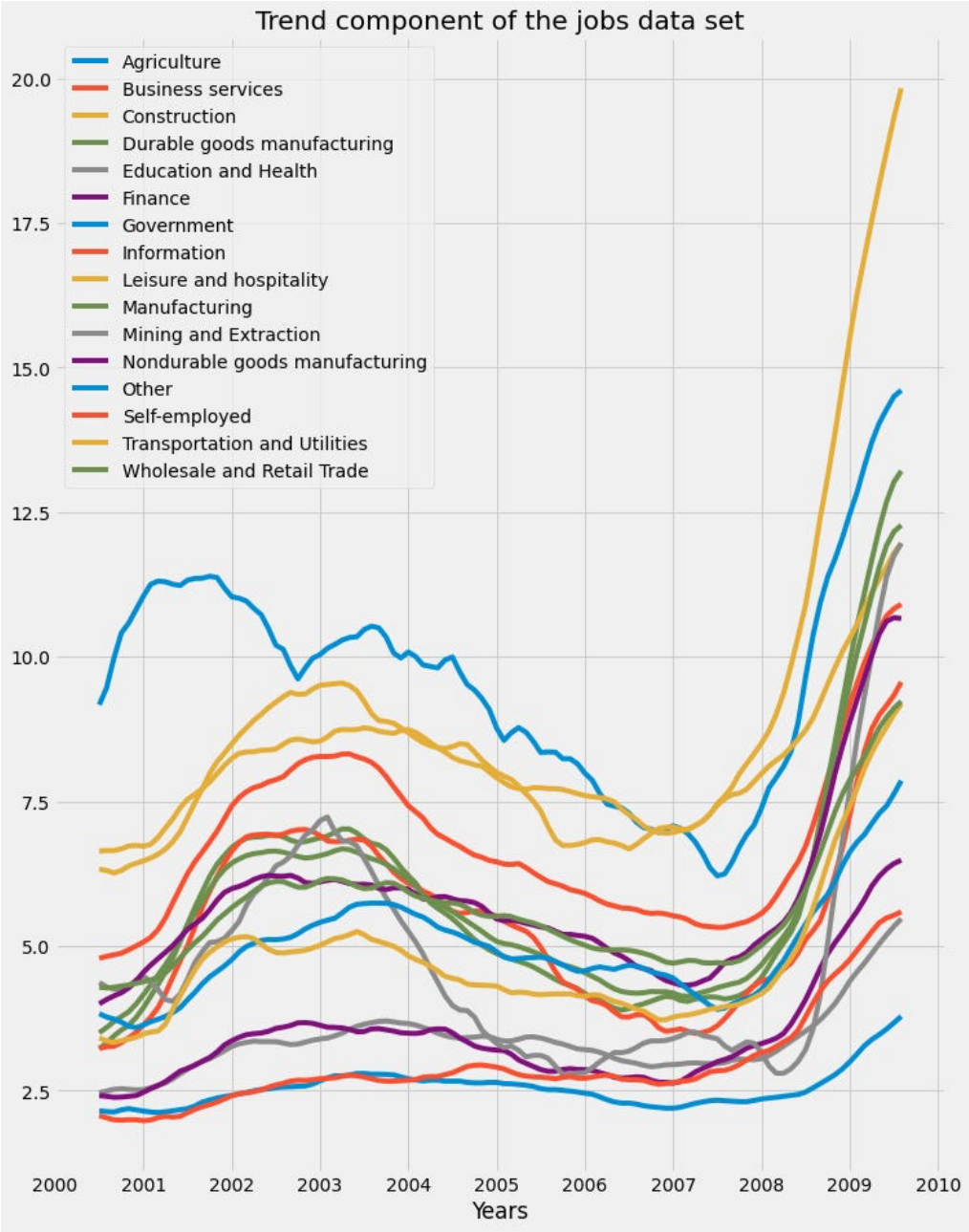


Figure 3.38. The trend component of the job's dataset

We can see how the 2008 financial crisis led to a rise in unemployment rates across all industries. Finally, the residual component of the jobs dataset is plotted:

Figure 3.39. Noise component of the job's dataset.

5.3. Compute the correlations between the time series of the jobs dataset

First, compute the correlation between all columns in the **trend_df** DataFrame using the Spearman method and assign the results to a new variable called **trend_corr**.

Then, generate a **clustermap()** of the correlation matrix trend_corr by using the clustermap function from the seaborn library. Lines 3 and 4 specify a rotation angle of 0 to the y-axis labels and a rotation angle of 90 to the x-axis labels.

```
# Plotting a clustermap of the jobs correlation matrix

# Get correlation matrix of the seasonality_df DataFrame
trend_corr = trend_df.corr(method='spearman')

# Customize the clustermap of the seasonality_corr
fig = sns.clustermap(trend_corr, annot=True,
linewidth=0.4,figsize=(15,10))

plt.setp(fig.ax_heatmap.yaxis.get_majorticklabels(), rotation=0)
plt.setp(fig.ax_heatmap.xaxis.get_majorticklabels(), rotation=90)
```

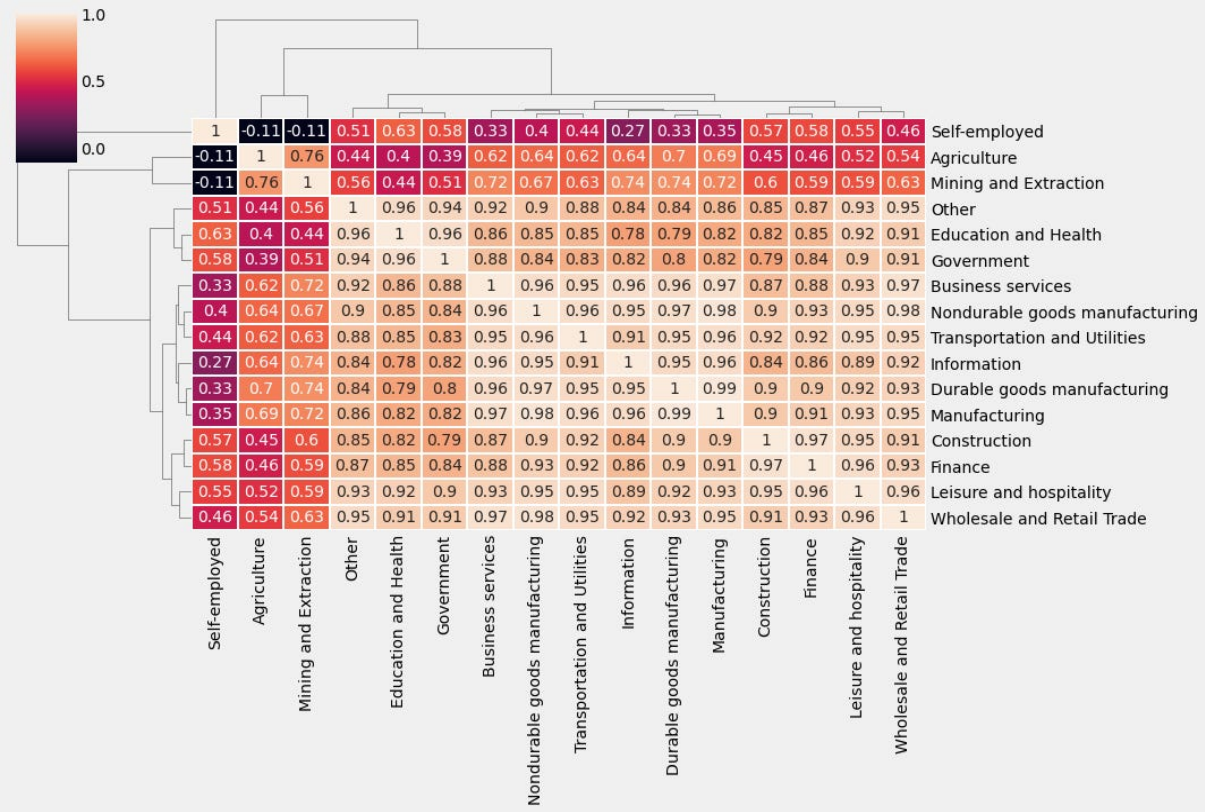


Figure 3.40. Cluster map of the job's dataset correlation matrix.

Time Series Forecasting with ARIMA Models

Table of Contents:

Up until this point, our journey has focused on looking backward—cleaning, manipulating, and visualizing historical data to understand what has happened. In this chapter, we cross the threshold into the most exciting domain of time series analysis: forecasting. We stop asking "what happened?" and start answering "what happens next?"

This chapter introduces you to the gold standard of statistical forecasting: the ARIMA family of models. We will begin by tackling the fundamental concept of stationarity—the idea that to predict the future, the statistical properties of our data must remain constant over time. You will learn to diagnose non-stationary data using the Augmented Dickey-Fuller test and transform it using differencing.

From there, we will build up from simple Autoregressive (AR) and Moving Average (MA) models to the powerful ARIMA (Autoregressive Integrated Moving Average) and SARIMA (Seasonal ARIMA) architectures. We will not just fit models blindly; we will adopt the rigorous Box-Jenkins method, a systematic workflow for identifying parameters using ACF/PACF plots and Information Criteria (AIC/BIC), estimating coefficients, and diagnosing residuals. By the end of this chapter, you will be able to build robust models capable of capturing complex seasonal patterns and automating parameter searches to forecast everything from candy production to CO2 emissions.

1. ARMA Models

We will start with a small introduction to stationarity and how this is important for ARMA models. Then we will revise how to test for stationarity by eye and with a standard statistical test. If you would like to get more information about these topics, you can check chapter 2, **Time Series Analysis in Python**, as I have covered in more detail in it. Finally, you'll learn the basic structure of ARMA models and use this to generate some ARMA data and fit an ARMA model.

We will use the **candy production dataset**, which represents the monthly candy production in the US between 1972 and 2018. Specifically, we will be using the industrial production index IPG3113N. This is the total amount of sugar and confectionery products produced in the USA

per month, as a percentage of the January 2012 production. So 120 would be 120% of the January 2012 industrial production.

1.1. Introduction to stationarity

Stationary means that the distribution of the data doesn't change with time. For a time series to be stationary, it must fulfill three criteria:

- **The series has zero trends**. It isn't growing or shrinking.
- **The variance is constant**. The average distance of the data points from the zero line isn't changing.
- **The autocorrelation is constant**. How each value in the time series is related to its neighbors stays the same.

The importance of stationarity comes from that to model a time series, it must be stationary. The reason for this is that modeling is all about estimating parameters that represent the data; therefore, if the parameters of the data are changing with time, it will be difficult to estimate all the parameters. Let's first load and plot the monthly candy production dataset:

```
# Load in the time series
candy = pd.read_csv('candy_production.csv',
            index_col='date',
            parse_dates=True)
# change the plot style into fivethirtyeight
plt.style.use('fivethirtyeight')

# Plot and show the time series on axis ax1
fig, ax1 = plt.subplots()
candy.plot(ax=ax1, figsize=(12,10))
plt.show()
```

Generally, in machine learning, you have a training set on which you fit your model, and a test set on which you will test your predictions against. Time series forecasting is just the same. Our train-test split will be different.

We use the past values to make future predictions, and so we will need to split the data in time. We train on the data earlier in the time series and test on the data that comes later. We can split a time series at a given date as shown below using the DataFrame's **.loc** method.

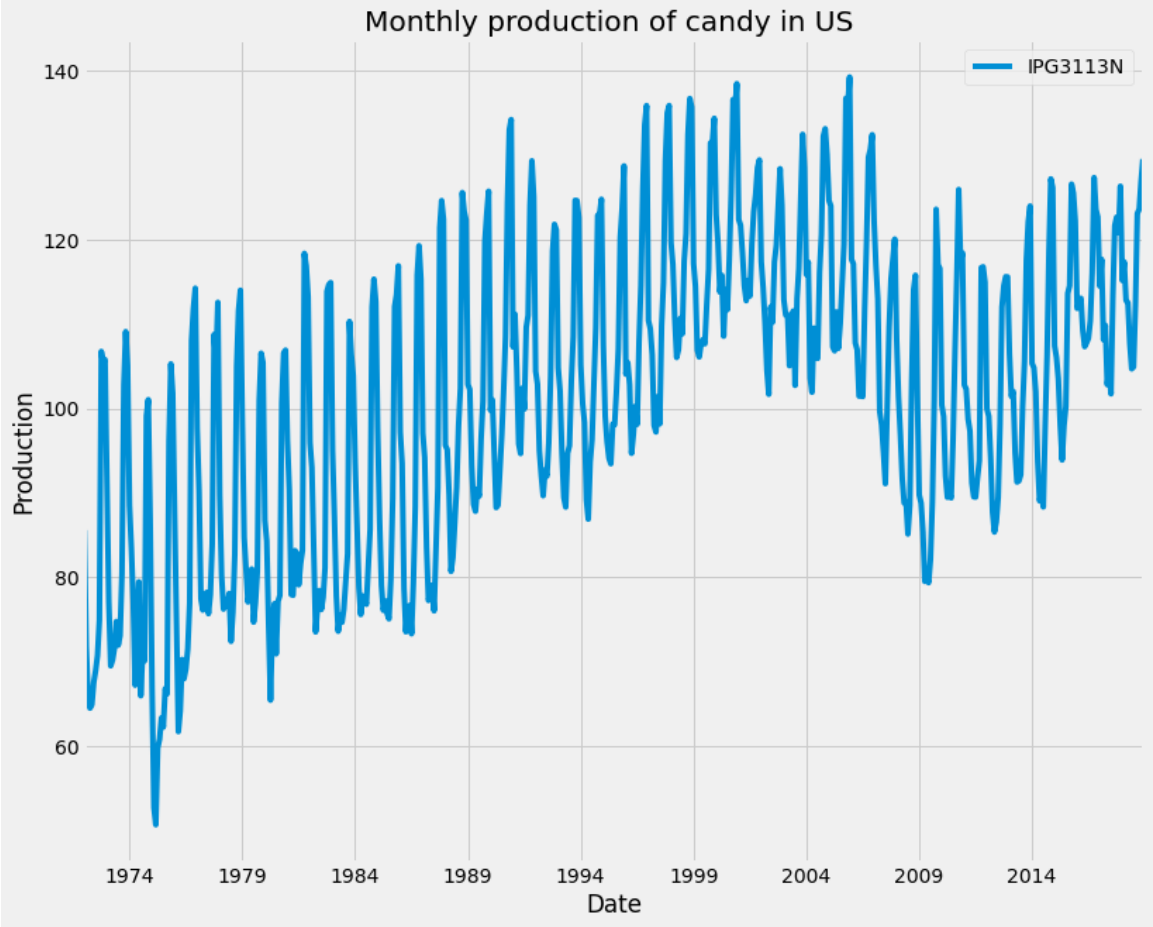


Figure 4.1. Monthly production of candy in the US from 1974 to 2018.

```
# Split the data into a train and test set
candy_train = candy.loc[:'2006']
candy_test = candy.loc['2007':]

# Create an axis
fig, ax = plt.subplots()

# Plot the train and test sets on the axis ax
candy_train.plot(ax=ax, figsize=(12,10))
candy_test.plot(ax=ax)
plt.title('train - test split of the monthly production of candy in
US')
plt.xlabel('Date')
plt.ylabel('Production')
plt.show()
```

1.2. Making a time series stationary

There are many ways to test stationarity, one of them with the eyes, and others are more formal using statistical tests. There are also ways to transform non-stationary time series into stationary ones. We'll address both of these in this subsection, and then you'll be ready to start modeling. The most common test for identifying whether a time series is non-stationary is the augmented Dicky-Fuller test. This is a statistical test, where the null hypothesis is that your time series is non-stationary due to trends.

We can implement the augmented Dicky-Fuller test using statsmodels. First, we import the **adfuller** function as shown, then we can run it on the candy production time series.

```
from statsmodels.tsa.stattools import adfuller
results = adfuller(candy)
print(results)
```

```
(-1.7760153075016107, 0.3924327500714, 14, 549, {'1%': -3.4423174665535385, '5%': -2.866818952732754,
'10%': -2.569581505602171}, 3094.828881317046)
```

The results object is a tuple. The zeroth element is the test statistic; in this case, it is -1.77. The more negative this number is, the more likely that the data is stationary. The next item in the results tuple is the test p-value. Here it's 0.3. If the p-value is smaller than 0.05, we reject the null hypothesis and assume our time series must be stationary.

The last item in the tuple is a dictionary. This stores the critical values of the test statistic, which equate to different p-values. In this case, if we wanted a p-value of 0.05 or below, our test statistic needed to be below -2.86.

Based on this result, we are sure that the time series is non-stationary. Therefore, we will need to transform the data into a stationary form before we can model it.
We can think of this a bit like feature engineering in classic machine learning. One very common way to make a time series stationary is to take its difference. This is where from each value in our time series we subtract the previous value.

```
# Calculate the first difference and drop the nans
candy_diff = candy.diff()
candy_diff = candy_diff.dropna()

# Run test and print
result_diff = adfuller(candy_diff)
print(result_diff)
```

```
(-6.175912489755653, 6.631549159335702e-08, 13, 549, {'1%': -3.4423174665535385, '5%': -2.86681895273
2754, '10%': -2.569581505602171}, 3091.3123583091397)
```

From the results, we can see that now the time series are stationary. This time, taking the difference was enough to make it stationary, but for other time series, we may need to make the difference more than once or do other transformations.

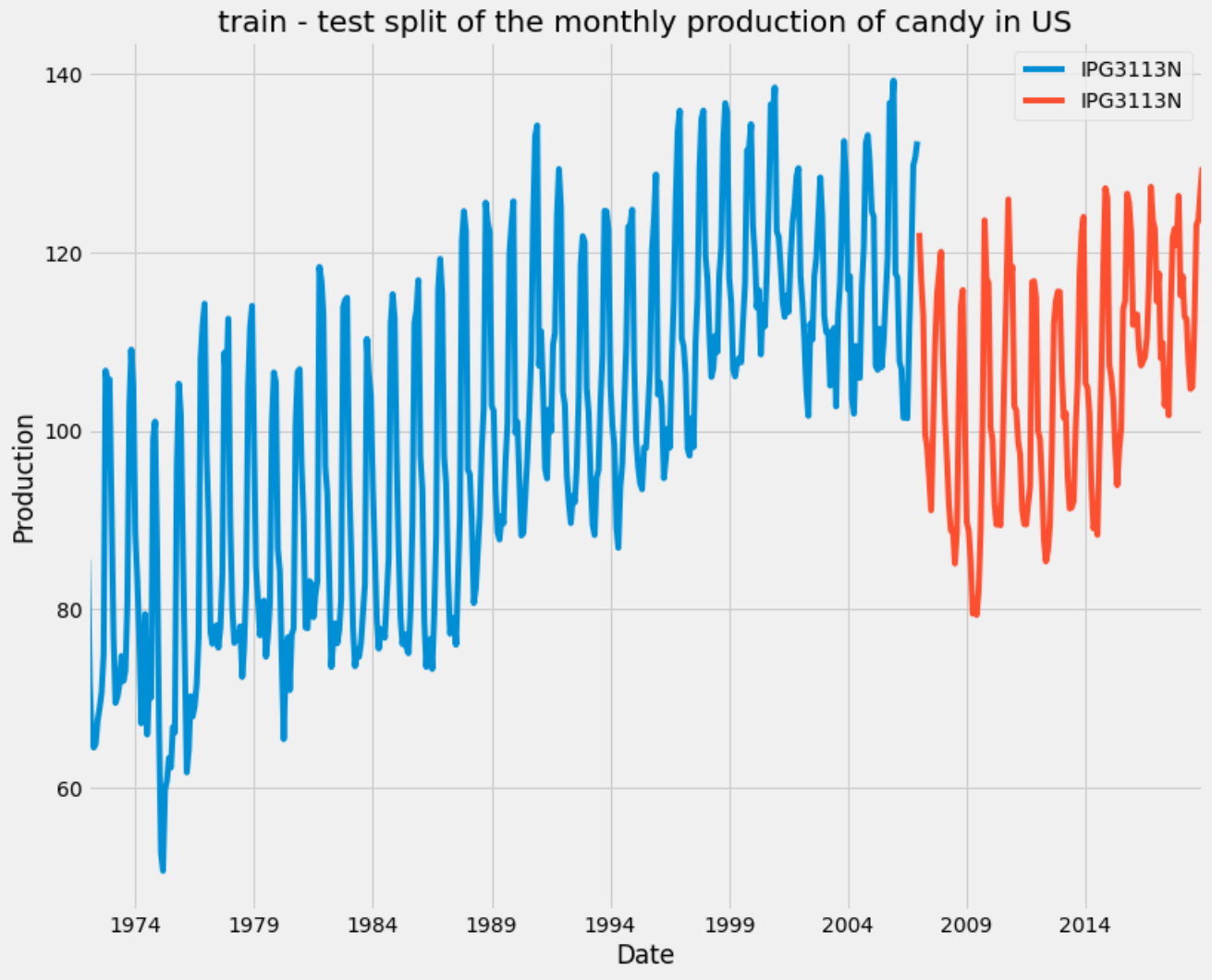


Figure 4.2. Train—test split of the monthly production of candy in the US.

Sometimes we will need to perform other transformations to make the time series stationary. This could be to take the log, or the square root of a time series, or to calculate the proportional change. It can be hard to decide which of these to do, but often the simplest solution is the best one.

1.3. Introduction to AR, MA, and ARMA models

In an autoregressive (AR) model, we regress the values of the time series against previous values of the same time series. The equation for a simple AR model is shown below:

$$y(t) = a(1) \times y(t-1) + \in (t)$$

The value of the time series at time (t) is the value of the time series at the previous step multiplied by parameter a(1), added to a noise or shock term ϵ(t). The shock term is white noise, meaning each shock is random and not related to the other shocks in the series.
The a(1) is the autoregressive coefficient at lag one. Compare this to a simple linear regression where the dependent variable is y(t) and the independent variable is y(t-1). The coefficient a(1) is just the slope of the line, and the shocks are the residuals of the line.

This is a first-order AR model. The order of the model is the number of time lags used. An order two AR model has two autoregressive coefficients and has two independent variables, the series at lag one and the series at lag two. More generally, we use p to mean the order of the AR model. This means we have p autoregressive coefficients and use p lags.

In a **moving average (MA)** model, we regress the values of the time series against the previous shock values of this same time series. The equation for a simple MA model is shown below:

$$y(t) = m(1) \times \in (t-1) + \in (t)$$

The value of the time series y(t)is m(1) times the value of the shock at the previous step, plus a shocking term for the current time step. This is a first-order MA model. Again, the order of the model means how many time lags we use. An MA two model would include shocks from one and two steps ago. More generally, we use q to mean the order of the MA model.

An ARMA model is a combination of the AR and MA models. The time series is regressed on the previous values and the previous shock terms. This is an ARMA-one-one model.
More generally, we use **ARMA(p,q)** to define an ARMA model. The p tells us the order of the autoregressive part of the model, and the q tells us the order of the moving average part.

$$y(t) = a(1) \times y(t-1) + m(1) \times \in (t-1) + \in (t)$$

Using the **statsmodels** package, we can both fit ARMA models and create ARMA data. Let's take this ARMA-one-one model. Say we want to simulate data with these coefficients.
First, we import the arma-generate-sample function. Then we make lists for the AR and MA coefficients. Note that both coefficient lists start with one. This is for the zero-lag term, and we will always set this to one. We set the lag one AR coefficient as 0.5 and the MA coefficient as 0.2.

We generate the data, passing in the coefficients, the number of data points to create, and the standard deviation of the shocks. **Here, we actually pass in the negative of the AR coefficients we desire. This is a quirk we will need to remember.**

```
from statsmodels.tsa.arima_process import arma_generate_sample
ar_coefs = [1, -0.5]
ma_coefs = [1, 0.2]
y = arma_generate_sample(ar_coefs, ma_coefs, nsample=100, scale=0.5)
```

The generated data can be represented with this equation:

$$y(t) = 0.5 \times y(t-1) + 0.2 \times \in (t-1) + \in (t)$$

Fitting is covered in the next section, but here is a quick peek at how we might fit this data. First, we import the ARMA model class. We instantiate the model, feed it the data, and define the model order. Then, finally, we fit.

```
from statsmodels.tsa.arima.model import ARIMA

# Instantiate model object
model = ARIMA(y, order=(1,0,1))

# Fit the model
results = model.fit()
```

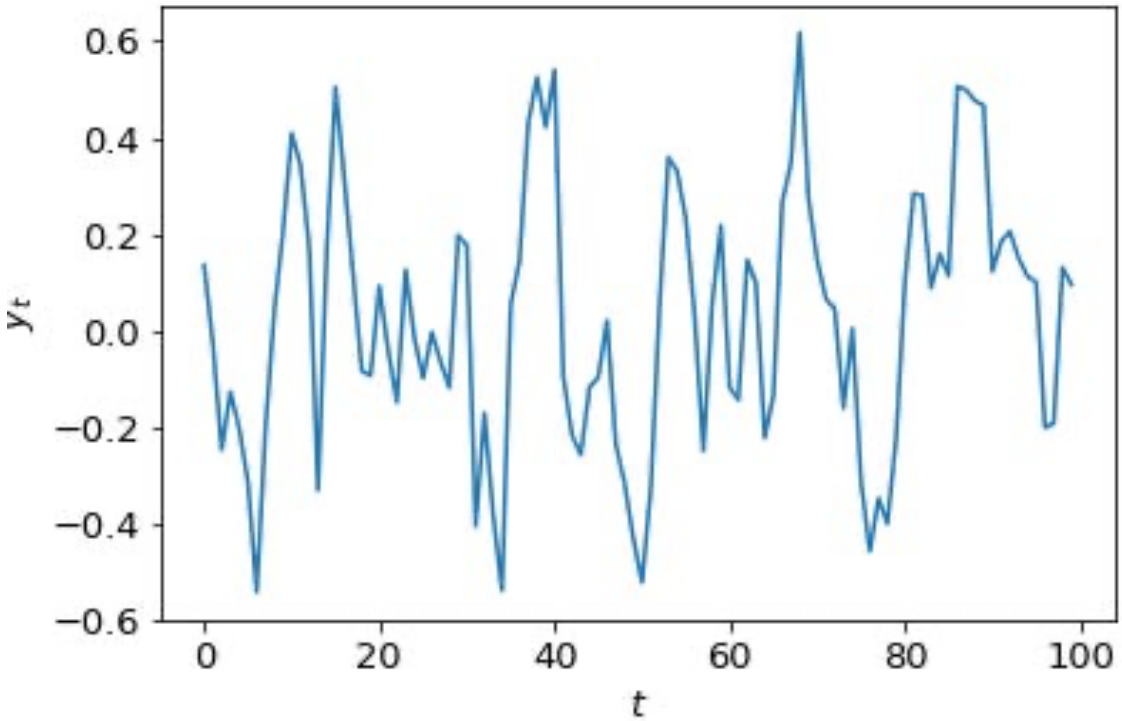


Figure 4.3.. The generated data with the ARMA model.

2. Fitting the Future

In this section, you'll learn how to use the elegant statsmodels package to fit ARMA, ARIMA, and ARMAX models. Then you'll use your models to predict the uncertain future of **Amazon stock prices**.

2.1. Fitting time series models

We had a quick look at fitting time series models in the last section, but let's have a closer look.
To fit these models, we first import the **ARIMA** model class from the **statsmodels** package.
We create a model object and define the model order; we must also feed in the training data. The data can be a pandas dataframe, a pandas series, or a NumPy array.

Remember that the order for an ARIMA model is (p,d,q) p is the autoregressive lags, d is the order of the difference, and q is the moving average lags. d is always an integer, while p and q may either be integers or lists of integers.
To fit an AR model, we can simply use the ARMA class with q equal to zero. To fit an MA model, we set p equal to zero. Let's have a look at the result summary of the fitted model :

```
from statsmodels.tsa.arima.model import ARIMA

# Instantiate model object
model = ARIMA(y, order=(1,0,1))

# Fit model
results = model.fit()
print(results.summary())
```

```
                               SARIMAX Results
==============================================================================
Dep. Variable:                      y   No. Observations:                  100
Model:                 ARIMA(1, 0, 1)   Log Likelihood                -66.653
Date:                Sun, 10 Apr 2022   AIC                            141.305
Time:                        18:01:42   BIC                            151.726
Sample:                             0   HQIC                           145.523
                                - 100
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
const          0.1195      0.145      0.827      0.408      -0.164       0.403
ar.L1          0.6481      0.114      5.668      0.000       0.424       0.872
ma.L1          0.0802      0.159      0.503      0.615      -0.232       0.393
sigma2         0.2206      0.031      7.060      0.000       0.159       0.282
===================================================================================
Ljung-Box (L1) (Q):                   0.01   Jarque-Bera (JB):                 0.07
Prob(Q):                              0.92   Prob(JB):                         0.97
Heteroskedasticity (H):               1.07   Skew:                             0.05
Prob(H) (two-sided):                  0.86   Kurtosis:                         3.07
===================================================================================
```

The top section includes useful information such as the order of the model that we fit, the number of observations or data points, and the name of the time series. The S.D. of innovations is the standard deviation of the shock terms.

The next section of the summary shows the fitted model parameters. Here we fitted an ARIMA(1,0,1) model, so the model has AR-lag-1 and lag-1 coefficients. In the table, these are the ar.L1 and ma.L1 rows. The lag-1 MA coefficient is in the last row. The first column shows the model coefficients, whilst the second column shows the standard error in these coefficients. This is the uncertainty in the fitted coefficient values.

One possible extension to the ARMA model is to use exogenous inputs to create the **ARMAX model.** This means that we model the time series using other independent variables as well as the time series itself. This is like a combination of an ARMA model and a normal linear regression model.

The equations for two simple ARMA and ARMAX models are shown here.
The only difference is one extra term. We add a new independent variable z(t) multiplied by its coefficient x(1). Let's think of an example where ARMAX might be useful.

ARMA(1,1) model :

$$y(t) = a(1)y(t-1) + m(1) \times \in (t-1) + \in (t)$$

ARMAX(1,1) model :

$$y(t) = x(1) \times z(t) + a(1) \times y(t-1) \; + m(1) \in (t-1) + \in (t)$$

We can fit an **ARMAX model** using the same **ARMA model** class we used before. The only difference is that we will now feed in our exogenous variable using the exog keyword. The model order and the fitting procedure are just the same.

2.2. Forecasting

After introducing how to fit ARIMA models to data, let's see how to use them to forecast and predict the future. Let's take an example of a time series represented by an AR(1) model. At any time point in the time series, we can predict the next value by multiplying the previous value by the lag-one AR coefficient. If the previous value was 15 and the coefficient a-one is 0.5, we would estimate the next value is 7.5.

If the shock term had a standard deviation of 1, we would predict our lower and upper uncertainty limits to be 6.5 and 8.5. This type of prediction is called **one-step-ahead prediction**. Below is its equation:

$$y = 0.5x + \in (t)$$

2.3. ARIMA models for non-stationary time series

If the time series you are trying to forecast is non-stationary, you will not be able to apply the ARMA model to it. We first have to make the difference to make it stationary, and then we can use the ARMA model for it.

However, when we do this, we will have a model that is trained to predict the value of the difference of the time series. What we want to predict is not the difference, but the actual value

of the time series. We can achieve this by carefully transforming our prediction of the differences.

We start with predictions of the difference values. The opposite of taking the difference is taking the cumulative sum or integral. We will need to use this transform to go from predictions of the difference values to predictions of the absolute values.

We can do this using the **np.cumsum** function. If we apply this function, we now have a prediction of how much the time series changed from its initial value over the forecast period. To get an absolute value, we need to add the last value of the original time series to this.

```
from numpy import cumsum
mean_forecast = cumsum(diff_forecast) + df.iloc[-1, 0]
```

These steps of starting with non-stationary data, differencing to make it stationary, and then integrating the forecast are very common in time series modeling. This is a lot of work! But thankfully, there is an extension of the ARMA model which does it for us! This is the autoregressive integrated moving average model (**ARIMA**).

We can implement an ARIMA model using the SARIMAX model class from statsmodels. The ARIMA model has three model orders. These are **p** the autoregressive order, **d** the order of differencing, and **q** the moving average order. In the previous section, we were setting the middle order parameter **d** to zero. If **d** is zero, we simply have an ARMA model.

When we use this model, we pass it a non-differenced time series and the model order. When we want to differentiate the time series data just once and then apply an ARMA(2,1) model. This is achieved by using an ARIMA(2,1,1) model. After we have stated the difference parameter, we don't need to worry about differencing anymore. We fit the model as before and make forecasts. The differencing and integration steps are all taken care of by the model object. This is a much easier way to get a forecast for non-stationary time series!

We must still be careful about selecting the right amount of differencing. Remember, we differ our data only until it is stationary and no more. We will work this out before we apply our model, using the augmented Dicky-Fuller test to decide the difference order. So by the time we come to apply a model, we already know the degree of differencing we should apply.

Let's apply this to real data. The data that will be used is the **Amazon stock price data.** We will apply the two methods. First, using ARMA models on the data with a difference, and using the ARIMA model with a built-in difference.

First, the data will be uploaded and plotted in Figure 4.4.

```
amazon = pd.read_csv('amazon_close.csv',
                     index_col='date',
                     parse_dates=True)
amazon.plot()
plt.title('Amazon stock price change with time')
```

```
plt.ylabel('Stock price')
```

First, we will apply the Adfuller-Dickey test to determine whether the time series is stationary or not.

```
from statsmodels.tsa.stattools import adfuller

# Run Dicky-Fuller test
result = adfuller(amazon)

# Print test statistic
print('The test stastics:', result[0])

# Print p-value
print("The p-value:",result[1])
```

```
The test stastics: -1.3446690965326022
The p-value: 0.6084966311408393
```

Figure 4.4. Amazon's stock price changes with time.

The p-value is bigger than 0.05; therefore, we cannot reject the null hypothesis, and the time series is considered to be non-stationary. Therefore, we will take the first difference and check whether this will make it stationary or not using also **Adfuller-Dickey** test.

```
# Take the first diff
amazon_diff = amazon.diff()
amazon_diff.dropna(inplace=True)
```

```
# Run Dicky-Fuller test
result = adfuller(amazon_diff)

# Print test statistic
print('The test stastics:', result[0])

# Print p-value
print("The p-value:",result[1])
```

```
The test stastics: -7.203579488811234
The p-value: 2.331271725487324e-10
```

The p-value after taking the first difference of the Amazon stock price time series is less than 0.05, so we can reject the null hypothesis, and the data is now considered stationary. For the modeling step, we can follow one of the two paths mentioned above.

First, use the ARMA model and apply it to the data with the first difference. Then the **np.cumsum** function will be used for the prediction of the actual data, not the data with the difference.

```
from statsmodels.tsa.arima.model import ARIMA

# Instantiate model object
model = ARIMA(amazon_diff, order=(1,0,1))

# Fit model
results = model.fit()
print(results.summary())
```

The second method is to use the ARIMA model and use the actual data, and use the difference parameter in the **ARIMA** function.

```
from statsmodels.tsa.arima.model import ARIMA

# Instantiate model object
model = ARIMA(amazon, order=(1,1,1))

# Fit model
results = model.fit()
print(results.suumary())
```

3. Finding the Best ARIMA Models

In this section, we will learn how to identify promising model orders from the data itself, then, once the most promising models have been trained, you'll learn how to choose the best model from this fitted selection. You'll also learn how to structure your time series project using the Box-Jenkins method.

3.1. Using ACF and PACF to find the best model parameters

In the previous section, we knew how to use the ARIMA model for forecasting; however, we did not discuss how to choose the order of the forecasting model.
The model order is a very important parameter that affects the quality of the forecasts. One of the best ways to identify the correct model order is the autocorrelation function (ACF) and partial autocorrelation function (PACF).

The **ACF** can be defined as the correlation between a time series and itself with n lags. So ACF(1) is the correlation between the time series and a one-step lagged version of itself.
An ACF(2) is the correlation between the time series and a one-step lagged version of itself, and so on. To have a better understanding of the ACF, let's plot it for the earthquake time series for ten lags.

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# Make ACF plot
plot_acf(earthquake['earthquakes_per_year'], lags=10, zero=False)
plt.show()
```

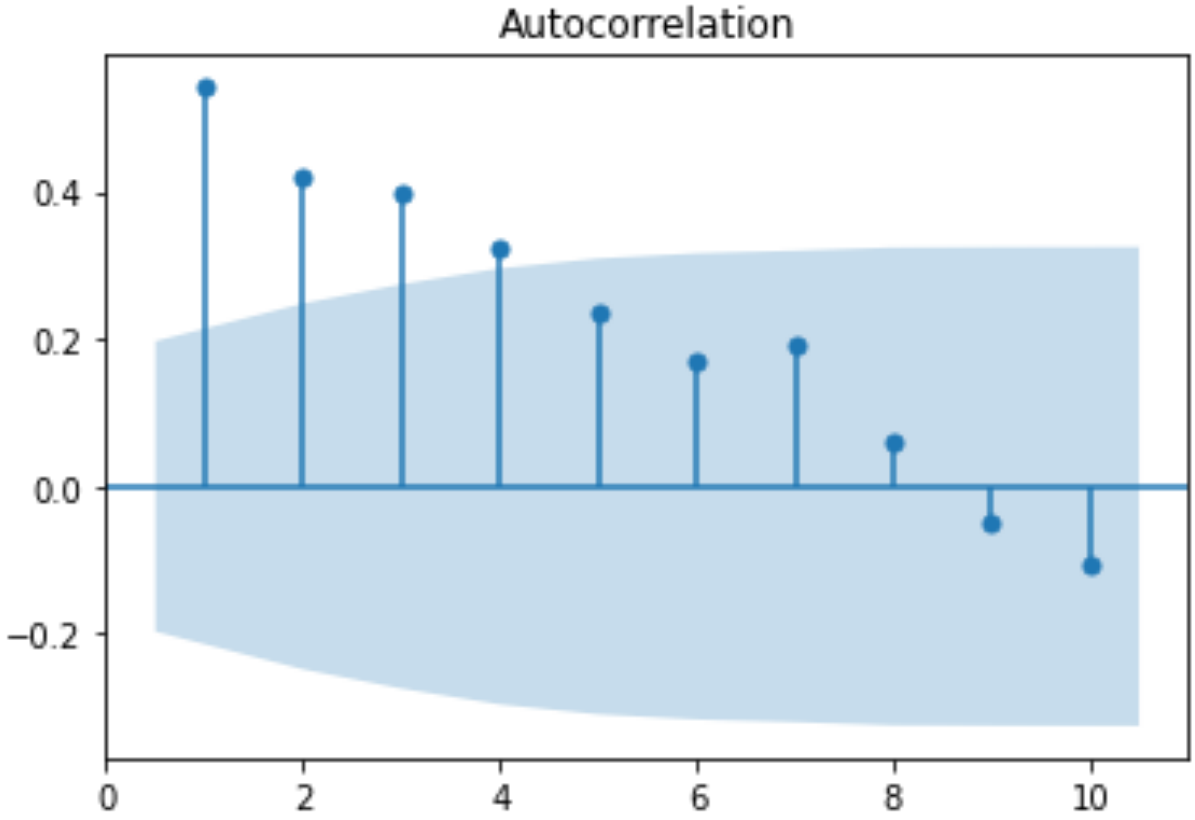


Figure 4.5. ACF of the earthquake time series.

The bars in the figure above show the correlation of the time series with itself at each lag. If the bar is inside the blue shaded region, this means that it is not statistically significant. On the other

hand, the **PACF** is the correlation between a time series and a lagged version of itself after subtracting the effect of correlation at smaller lags.

So it can be considered as the correlation at a particular lag. To have a better understanding of the PACF, let's plot it for the earthquake time series for ten lags.

```
# Make PACF plot
plot_pacf(earthquake['earthquakes_per_year'], lags=10, zero=False)
plt.show()
```

The bars in Figure 4.6 show the correlation of the time series with itself at each lag after subtracting the effect of the correlation of the smaller lags. If the bar is inside the blue shaded region, this means that it is not statistically significant.

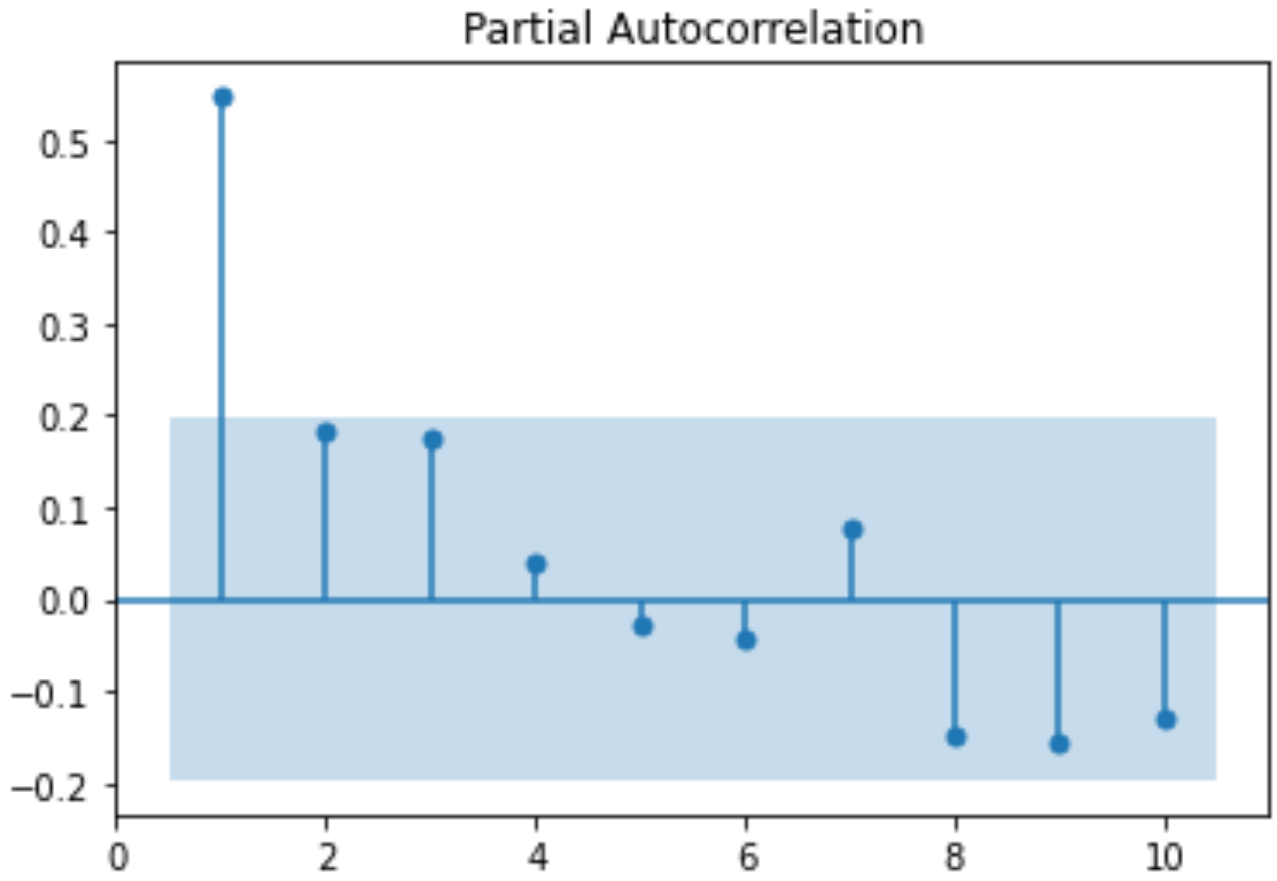


Figure 4.6.. PACF of the earthquake time series.

The time series must be made stationary before making these plots. If the ACF values are high and tail off very, very slowly, then this is a sign that the data is non-stationary, and it needs to be differenced. On the other hand, if the autocorrelation at lag-1 is very negative, this is a sign that we have taken the difference too many times.

By comparing the ACF and PACF for a time series, we can indicate the model order. There are three main possibilities:

- **AR(p) model:** ACF tails off and PACF cuts off after lag p. Figure 4.7 shows the ACF and PACF plots for an AR(2).

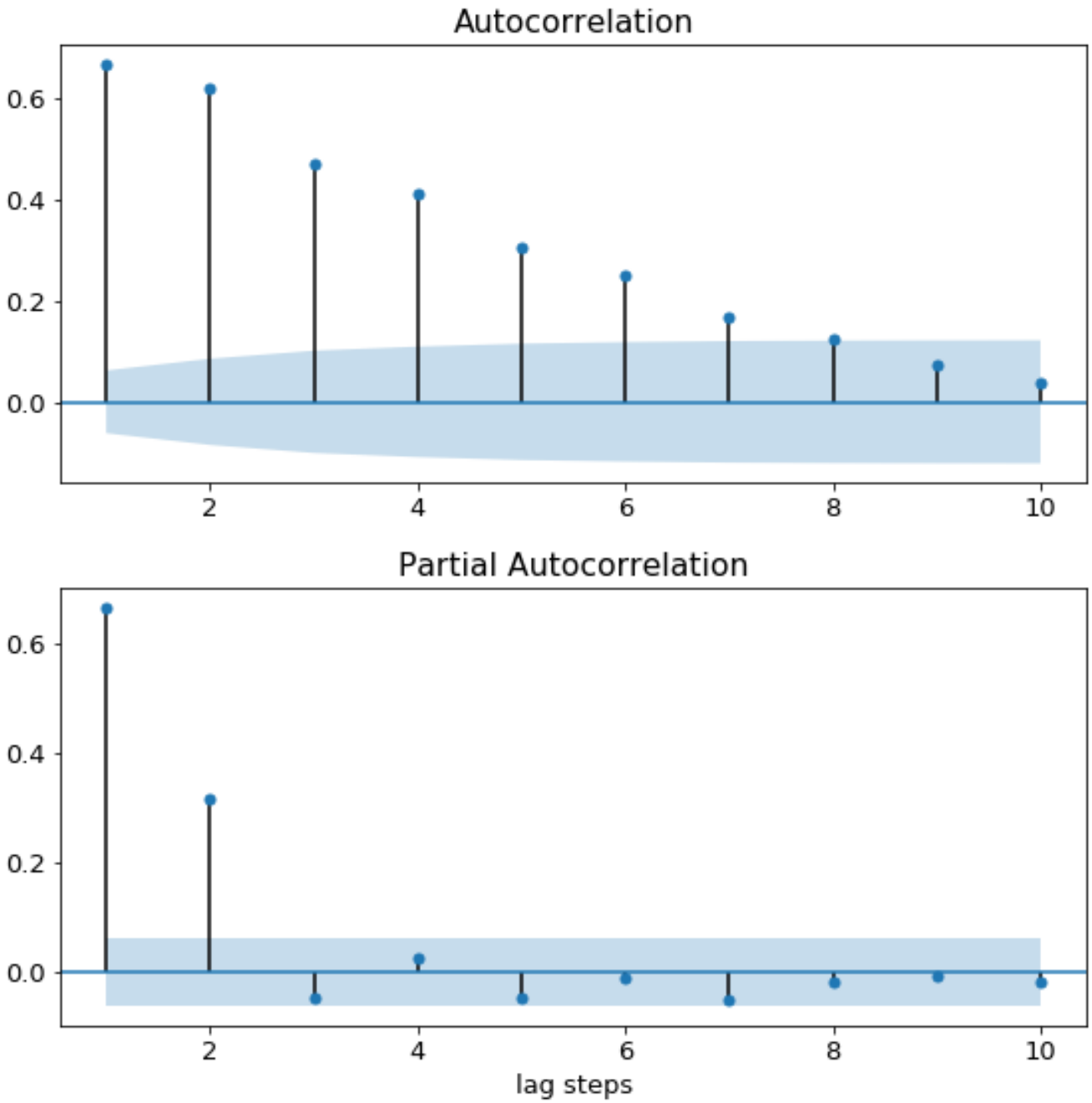


Figure 4.7. ACF and PACF plots for an AR (2).

- **MA(q) model:** The amplitude of the ACF cuts off after lag q, while the PACF tails off. Figure 4.8 shows the ACF and PACF plots for an MA(2).

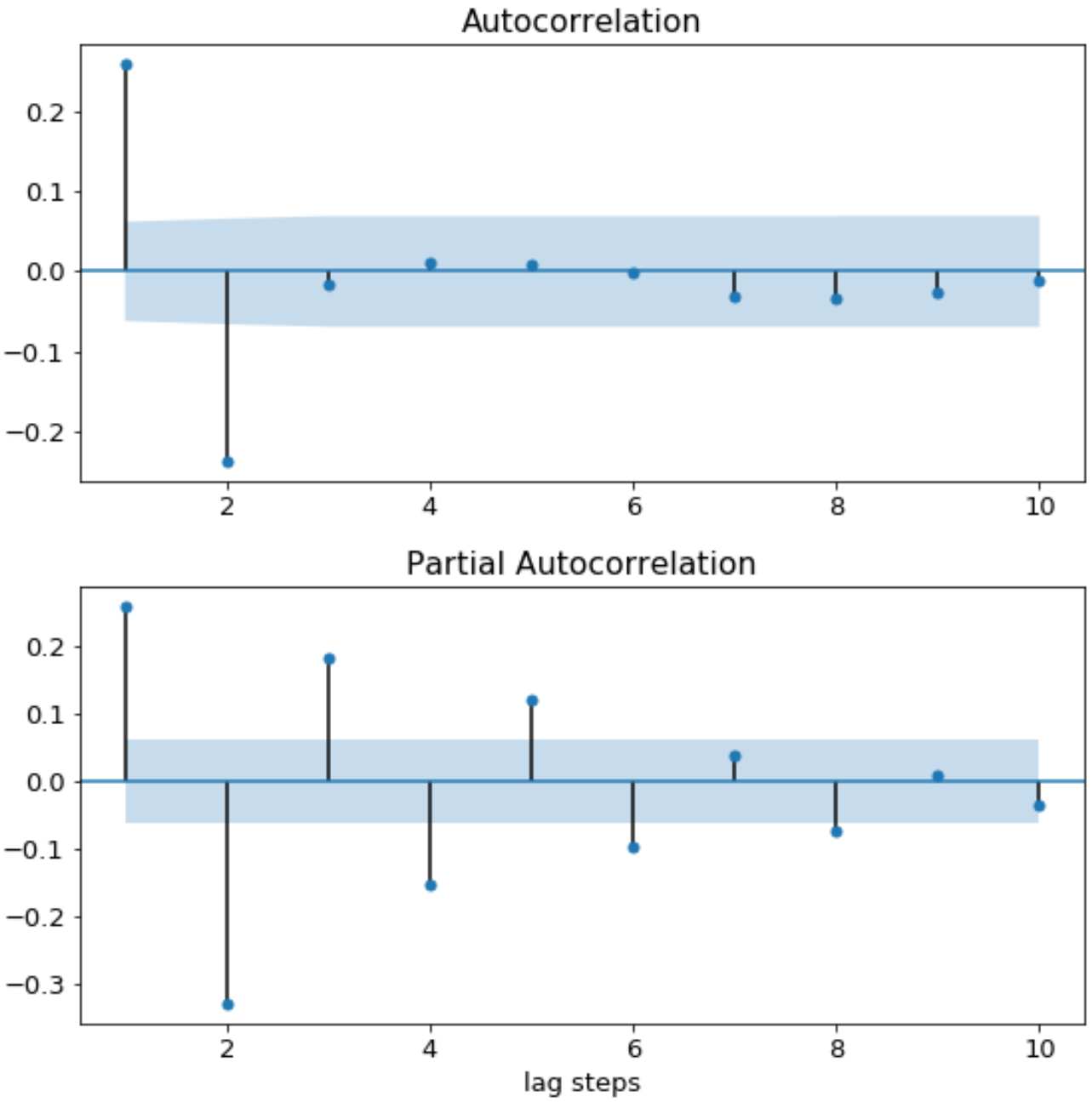


Figure 4.8. ACF and PACF plots for an MA (2).

- **ARIMA(p,q) model:** if the amplitude of both the ACF and PACF cuts off, then this will be the ARIMA model; however, we will not be able to deduce the model orders (p,q) from the plot.

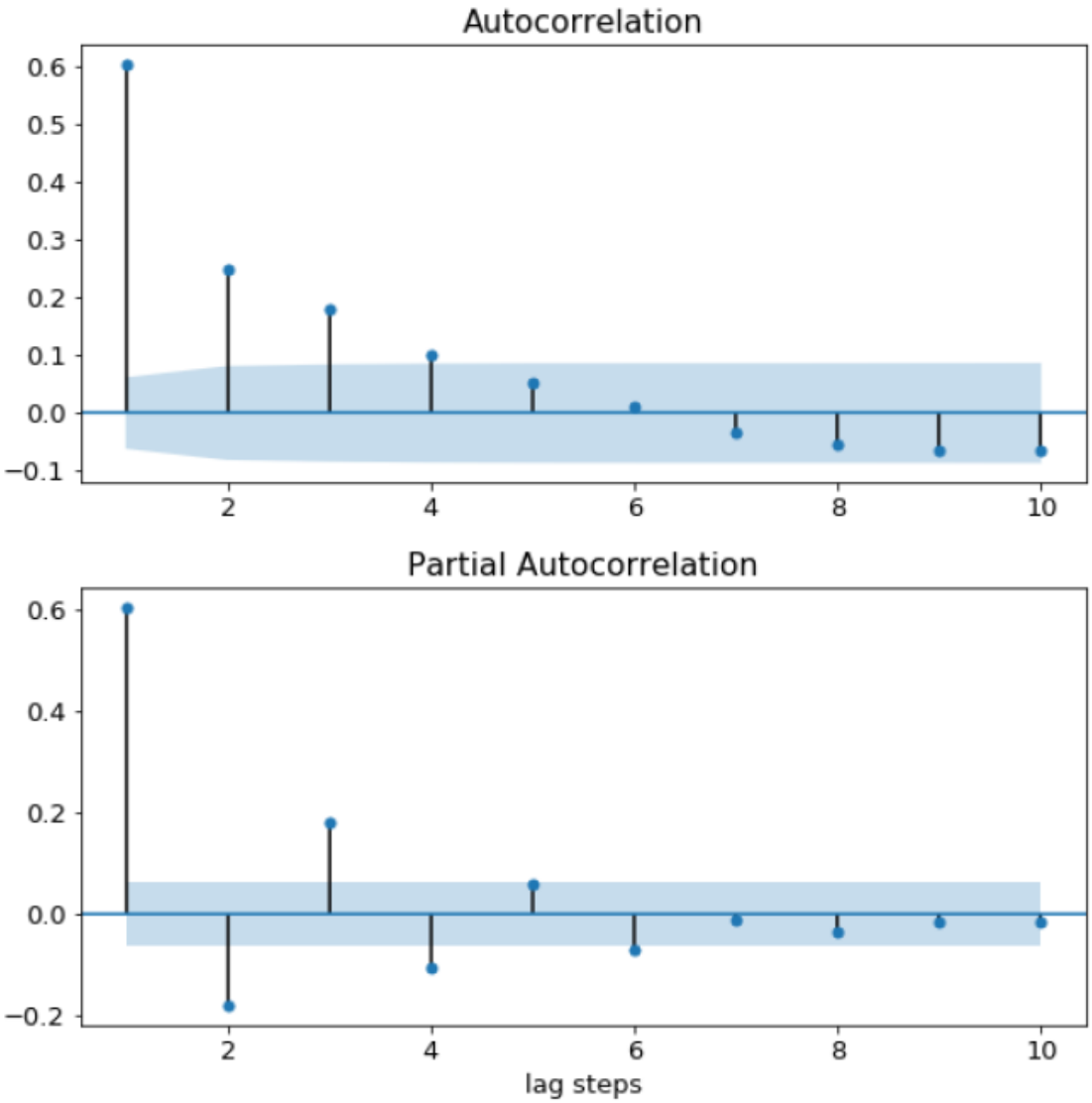


Figure 4.9. ACF and PACF plots for an ARIMA(p,q) model.

Let's apply this to the earthquake time series, which we plotted the ACF and PACF before. It looks like the ACF tails off while the PACF cuts off at a lag of 1. Therefore, it follows the first case, and the time series can be modeled using an AR(1) model.

3.2. Using AIC and BIC to narrow your model choices

We mentioned in the third case that when the amplitude of both the ACF and PACF cuts off, the time series can be modeled using an ARIMA(p,q) model. However, the order of the model (p,q)

can be indicated from the ACF and PACF plots. However, the Akaike information criterion (AIC) and Bayesian information criterion (BIC) can be used for finding their values.
For both **AIC** and **BIC**, lower values suggest a better model. Both AIC and BIC penalize complex models with a lot of parameters. However, they differ in the way they do so and how much they penalize the model complexity.

The **BIC** penalizes additional model orders more than **AIC,** and therefore, **BIC** will sometimes suggest a simpler model than AIC. They often will suggest the same model, but if not, you will have to choose one of them depending on your priorities.

If your goal is to identify a better predictive model, then AIC will be a better choice, and if your goal is to identify a good explanatory model, then BIC will be a better choice.
Let's apply this to the earthquake dataset used before and compare the results with the results we got from using ACF/PACF. We predict that the best model to fit the data is AR(1). We will calculate the AIC and BIC for different variations of p and q, starting from 0 to 2.

```
import statsmodels.api as sm
order_aic_bic =[]

# Loop over p values from 0-2
for p in range(3):
    # Loop over q values from 0-2
    for q in range(3):

        try:
            # create and fit ARMA(p,q) model
            model =
sm.tsa.statespace.SARIMAX(earthquake['earthquakes_per_year'],
order=(p, 0, q))
            results = model.fit()

            # Print order and results
            order_aic_bic.append((p, q, results.aic, results.bic))
        except:
            print(p, q, None, None)

# Make DataFrame of model order and AIC/BIC scores
order_df = pd.DataFrame(order_aic_bic, columns=['p', 'q',
'aic','bic'])

# let's sort them by AIC and BIC

# Sort by AIC
print("Sorted by AIC ")
print("\n")
print(order_df.sort_values('aic').reset_index(drop=True))

# Sort by BIC
print("Sorted by BIC ")
print("\n")
```

```
print(order_df.sort_values('bic').reset_index(drop=True))
```

The sorted AIC and BIC are shown below for each model.

```
Models sorted by AIC
```

	p	q	aic	bic
0	1	1	647.132300	654.917660
1	1	2	648.738566	659.119046
2	2	1	648.842840	659.223319
3	2	2	648.850644	661.826244
4	2	0	656.028374	663.813734
5	1	0	666.645526	671.835765
6	0	2	761.067479	768.852838
7	0	1	799.674173	804.864412
8	0	0	888.429772	891.024892

```
Models sorted by BIC
```

	p	q	aic	bic
0	1	1	647.132300	654.917660
1	1	2	648.738566	659.119046
2	2	1	648.842840	659.223319
3	2	2	648.850644	661.826244
4	2	0	656.028374	663.813734
5	1	0	666.645526	671.835765
6	0	2	761.067479	768.852838
7	0	1	799.674173	804.864412
8	0	0	888.429772	891.024892

The results show that the best model to fit the data is (1,1), which is different from what was found using the PACF/ACF.

3.3. The model diagnostic

The next step is to diagnose the model to know whether the model is behaving well or not. To diagnose the model, we will focus on the residuals of the training data. The residuals are the difference between the model's one-step-ahead predictions and the real values of the time series. In statsmodels, the residuals over the training period can be accessed using the dot-resid attribute of the results object. The results are stored as a pandas series. This is shown in the code below:

```
# The model with the best p and q found from the previous step
model = sm.tsa.statespace.SARIMAX(earthquake['earthquakes_per_year'],
order=(1, 0, 1))

# Fit model
results = model.fit()

# Assign residuals to variable
residuals = results.resid
print(residuals)
```

The residuals of the earthquake time series data were predicted with the ARIMA model of parameters (1,1).

```
date
1900-01-01    13.000000
1901-01-01     1.857276
1902-01-01    -5.045721
1903-01-01    -0.527074
1904-01-01     5.821641
                  ...
1994-01-01    -0.589256
1995-01-01     9.835545
1996-01-01     2.694023
1997-01-01    -0.278098
1998-01-01    -3.939243
Length: 99, dtype: float64
```

To know how far the predictions are from the true value, we can calculate the mean absolute error of the residuals. This is shown in the code below:

```
# The mean absolute error
mae = np.mean(np.abs(residuals))
print(mae)
```

The ideal model should have residuals that are uncorrelated white Gaussian noise centered on zero. We can use the results object's **.plot_diagnostics()** method to generate four common plots for evaluating this. The code for this is shown below:

```
# Create the 4 diagnostic plots
results.plot_diagnostics(figsize=(10,10))
plt.show()
```

There are four plots in the residuals diagnostic plots in Figure 4.10:

- **Standardized residuals plot:** The top left plot shows one-step-ahead standardized residuals. If our model is working correctly, there should be no obvious pattern in the residuals. This is shown here in this case.
- **Histogram plus estimated density plot:** This plot shows the distribution of the residuals. The histogram shows us the measured distribution; the orange line shows a smoothed version of this histogram, and the green line shows a normal distribution. If the model is good, these two lines should be the same. Here, there are small differences between them, which indicate that our model is doing well.
- **Normal Q-Q plot:** The Q-Q plot compares the distribution of the residuals to the normal distribution. If the distribution of the residuals is normal, then all the points should lie along the red line, except for some values at the end.
- **Correlogram plot:** The correlogram plot is the ACF plot of the residuals rather than the data. 95% of the correlations for lag greater than zero should not be significant (within the blue shades). If there is a significant correlation in the residuals, it means that there is information in the data that was not captured by the model.

We can get the summary statistics of the model residuals using the results **.summary()** method. In the result table shown below, the Prob(Q) is the p-value associated with the null hypothesis that the residuals have no correlation structure.

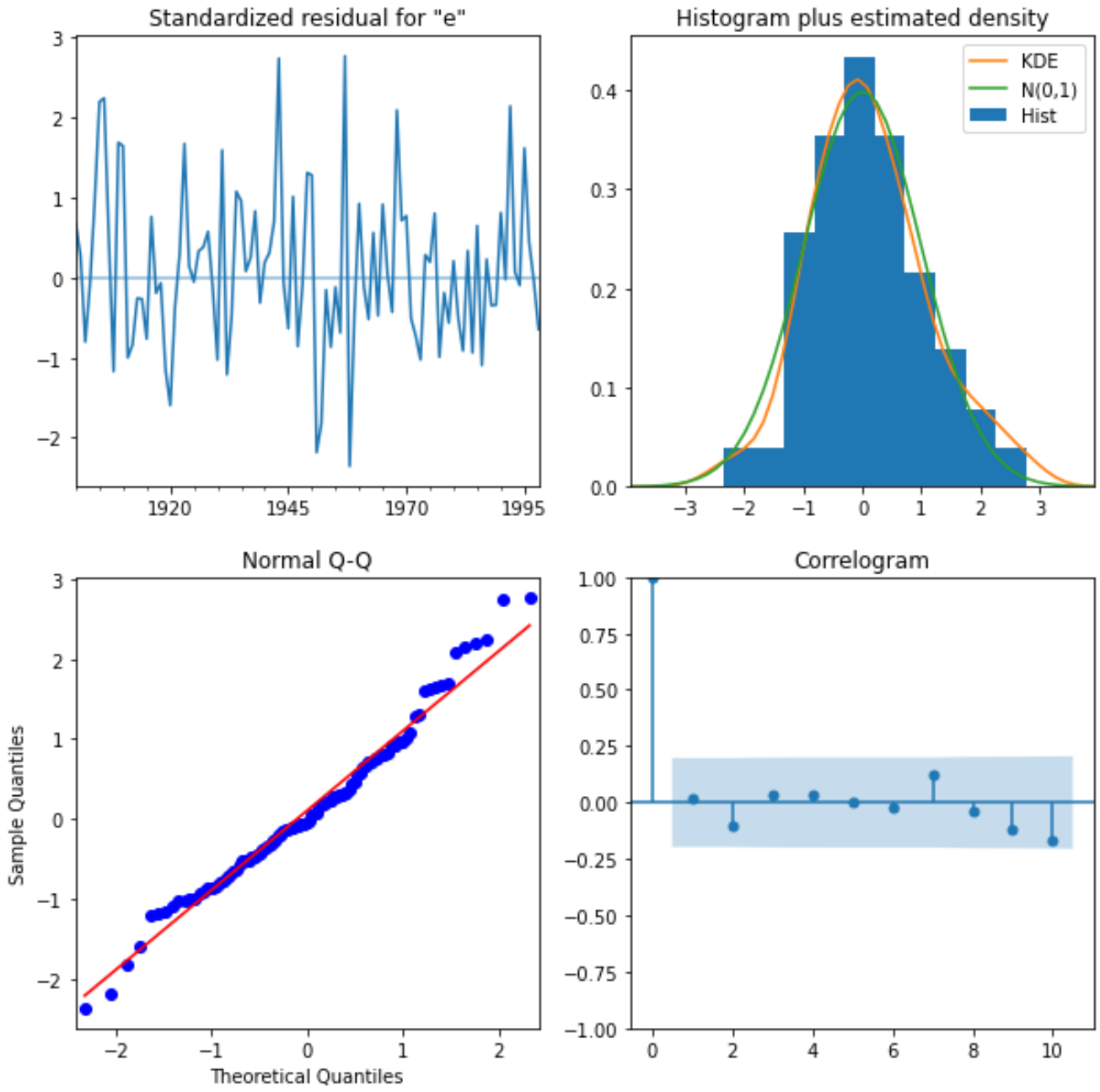


Figure 4.10. The residual diagnostic plots for the earthquake data.

Prob(JB) is the p-value associated with the null hypothesis that the residuals are normally distributed. If either of the p-values is less than 0.05, we reject that hypothesis.

```
# Summary statistics
print(results.summary())
```

```
                                SARIMAX Results
==========================================================================================
Dep. Variable:     earthquakes_per_year   No. Observations:                   99
Model:                 SARIMAX(1, 0, 1)   Log Likelihood                -320.566
Date:                  Sat, 30 Apr 2022   AIC                           647.132
Time:                          06:41:12   BIC                           654.918
Sample:                      01-01-1900   HQIC                          650.282
                           - 01-01-1998
Covariance Type:                   opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.9893      0.014     70.282      0.000       0.962       1.017
ma.L1         -0.5517      0.089     -6.198      0.000      -0.726      -0.377
sigma2        37.0254      4.980      7.434      0.000      27.264      46.787
===================================================================================
Ljung-Box (L1) (Q):                   0.05   Jarque-Bera (JB):                 2.76
Prob(Q):                              0.83   Prob(JB):                         0.25
Heteroskedasticity (H):               0.64   Skew:                             0.38
Prob(H) (two-sided):                  0.20   Kurtosis:                         3.31
===================================================================================
```

The values of Prob(Q) and Prob(JB) are both more than 0.05, so we cannot reject the null hypothesis that the residuals are **Gaussian** normally distributed.

3.4. The Box-Jenkins method

The Box-Jenkins method is a checklist that helps you go from raw data to a model ready for production. The three main steps in this method to follow to go from raw data to a production-ready model are identification, estimation, model diagnostics, and decision-making based on the model diagnostics. This could be summarized in Figure 4.11.

Identification

In the identification step, we explore and characterize the data to find the appropriate form of the data so as to be used in the estimation step. First, we will investigate whether the data is stationary or not. If the data is not stationary, we will find which transformation will transform the data into stationary data, and finally, we will identify the orders of p and q that are most promising.

The tools used in this step are plotting, augmented dicky-fuller test, differencing, and other transformations such as log transformation, and ACF and PACF for identifying promising model orders.

Estimation

In the estimation step, the model is trained and the AR and MA coefficients of the data. This is automatically done for us by the **model.fit()** method. In this step, we can fit many models and use BIC and AIC to narrow them down to more promising models.

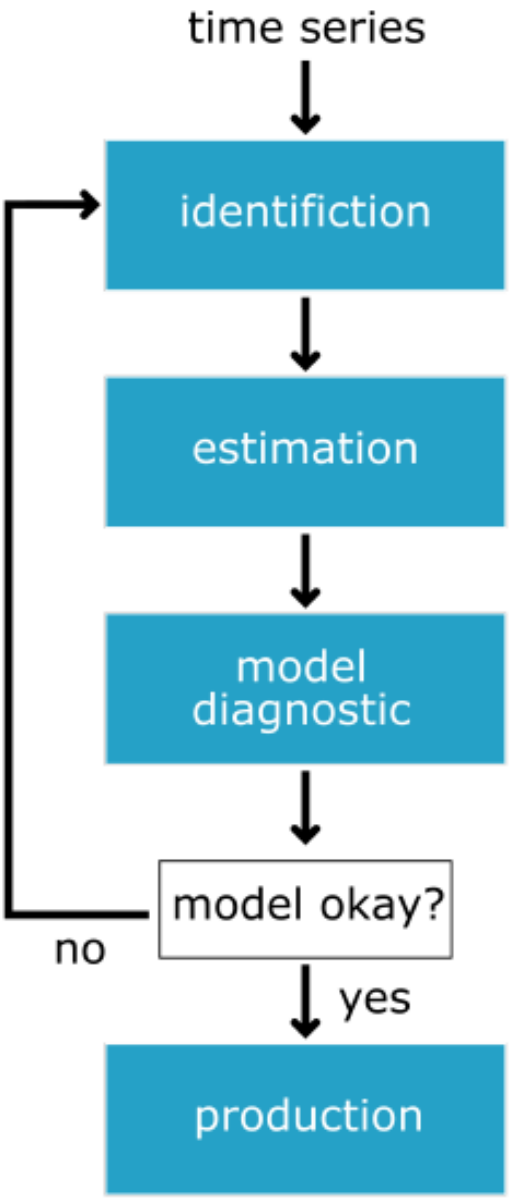


Figure 4.11.. The Box-Jenkins method allows you to go from raw time series data to a model ready for production.

Model Diagnostics

Using the information gathered from statistical tests and plots during the diagnostic step, we need to make a decision. Is the model good enough, or do we need to go back and rework it? If the residuals are not as they should be, we will go back and rethink our choices in the previous two steps. If the residuals are okay, then we can go ahead and make forecasts!

This should be your general project workflow when developing time series models. You may have to repeat the process a few times in order to build a model that fits well.

Let's apply the Box-Jenkins method to new data. The data will be used is the CO2 emission between 1958 to 2018. We will go through the three steps. We will start with identification. First, we will load and plot the data using the code below.

```
co2 = pd.read_csv('co2.csv', index_col='date', parse_dates=True)
co2.plot()
```

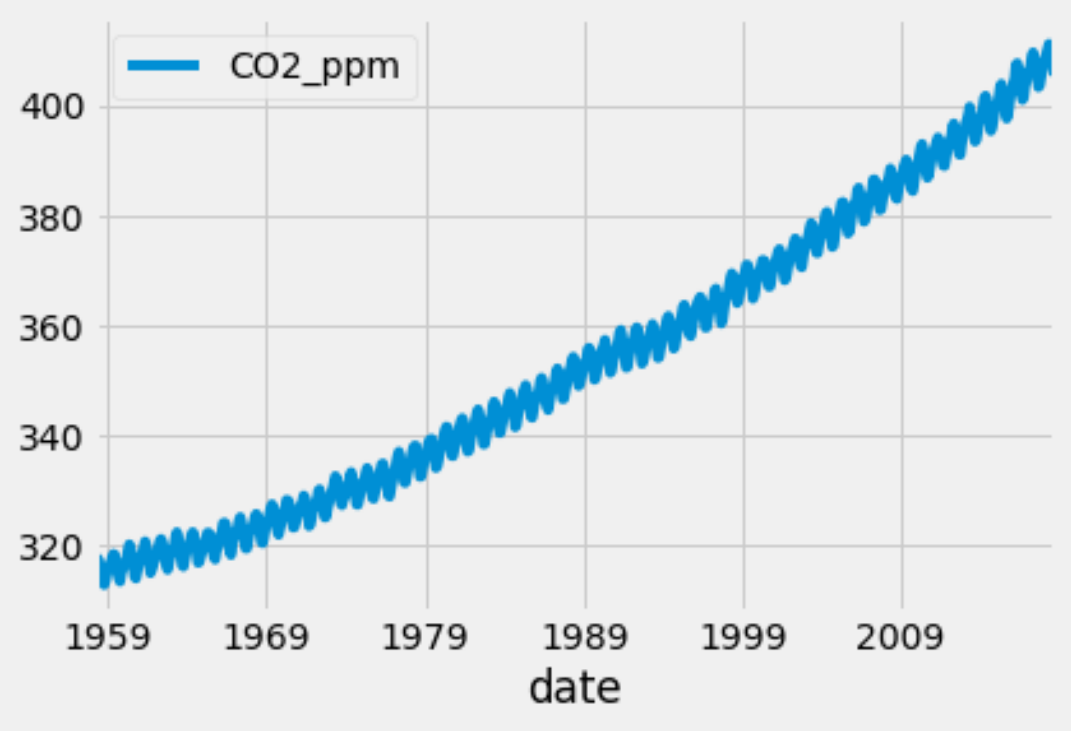


Figure 4.12. The CO2 emissions per year from 1959 to 2018.

There is a trend in the data. Then we will apply the augmented Dicky-Fuller test.

```
from statsmodels.tsa.stattools import adfuller

# Run Dicky-Fuller test
result = adfuller(co2)

# Print test statistic
print('The test stastics:', result[0])

# Print p-value
print("The p-value:",result[1])
```

```
The test stastics: 4.774866785205469
The p-value: 1.0
```

The p-value is 1, which means that we can not reject the null hypothesis that the time series is non-stationary. To convert it to stationery, let's take the first difference and plot it, and apply the augmented **Dicky-Fuller** test.

```
co2_diff = co2.diff()
co2_diff = co2_diff.dropna()
co2_diff.plot()

# Run Dicky-Fuller test
result_diff = adfuller(co2_diff)

# Print test statistic
print('The test stastics:', result_diff[0])

# Print p-value
print("The p-value:",result_diff[1])
```

```
The test stastics: -5.287425631615265
The p-value: 5.813789175332468e-06
```

The p-value is less than 0.05, so we can reject the null hypothesis and assume the data are stationary. The first difference in the CO2 data is shown in Figure 4.13.

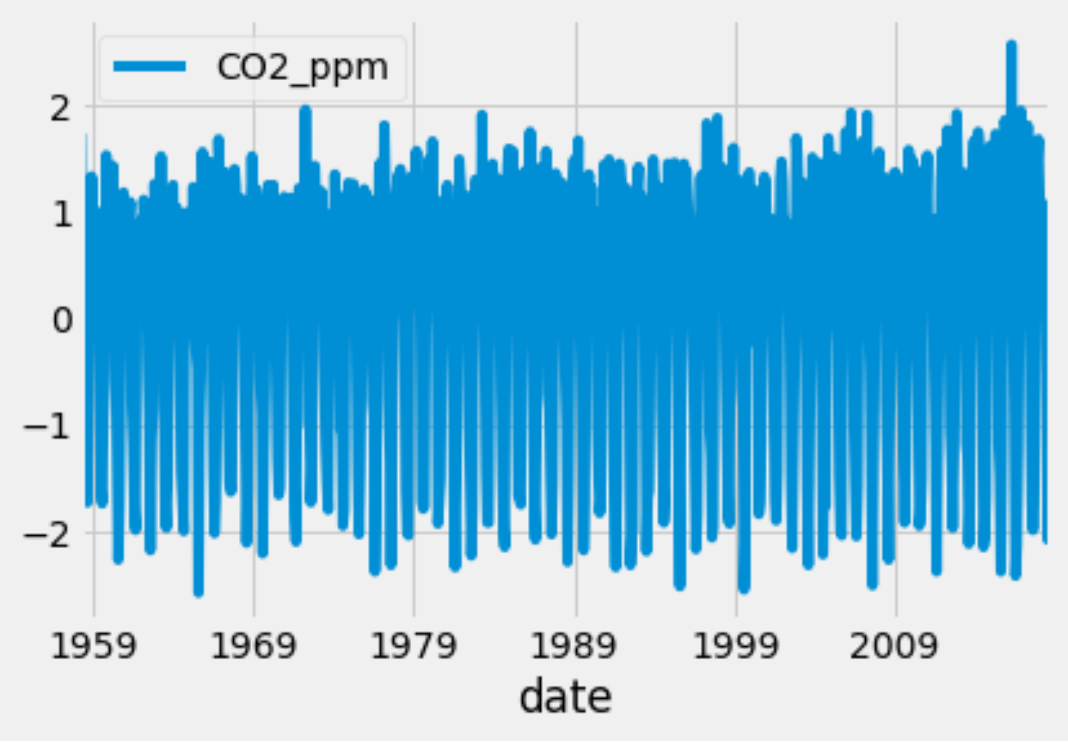


Figure 4.13. The first difference in the CO2 data.

The last step of the identification is to plot the ACF and PACF plots for the first difference in the CO2 emission data.

```
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# Create figure
fig, (ax1, ax2) = plt.subplots(2,1, figsize=(12,8))

# Plot the ACF of savings on ax1
plot_acf(co2_diff, lags=20, zero=False, ax=ax1)

# Plot the PACF of savings on ax2
plot_pacf(co2_diff, lags=20, zero=False, ax=ax2)

plt.show()
```

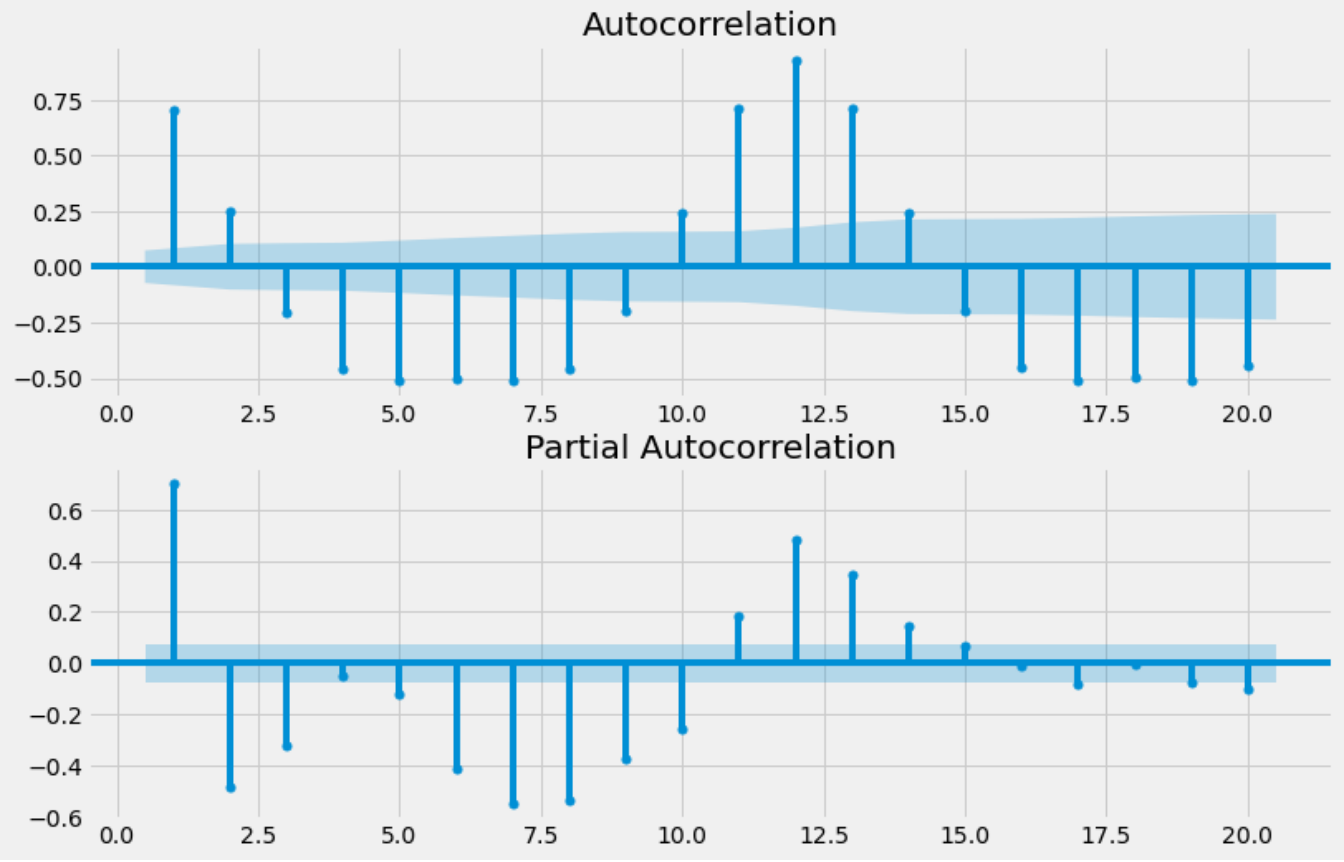


Figure 4.14. The ACF and PACF of the first difference of co2 emission data.

The ACF and PACF plots do not follow a certain pattern. So it will be difficult to identify the model order. So instead, we will use the AIC and BIC to narrow down the choice of the model order and then fit the data to the best model. This is the second step, which is the **estimation** step.

```
import statsmodels.api as sm
order_aic_bic =[]

# Loop over p values from 0-4
for p in range(5):
    # Loop over q values from 0-4
    for q in range(5):

        try:
            # create and fit ARMA(p,q) model
            model = sm.tsa.statespace.SARIMAX(co2, order=(p, 1, q))
            results = model.fit()

            # Print order and results
            order_aic_bic.append((p, q, results.aic, results.bic))
        except:
            print(p, q, None, None)

# Make DataFrame of model order and AIC/BIC scores
order_df = pd.DataFrame(order_aic_bic, columns=['p', 'q',
'aic','bic'])

# let's sort them by AIC and BIC

# Sort by AIC
print("Models sorted by AIC ")
print("\n")
print(order_df.sort_values('aic').reset_index(drop=True))

# Sort by BIC
print("Models sorted by BIC ")
print("\n")
print(order_df.sort_values('bic').reset_index(drop=True))
```

Below are the AIC and BIC for each model order.

```
Models sorted by AIC

     p  q           aic           bic
0    4  4    921.814711    963.102661
1    2  4   1297.058431   1329.171281
2    3  4   1300.818700   1337.519100
3    2  3   1525.544582   1553.069882
4    2  1   1526.197900   1544.548100
5    4  3   1526.885758   1563.586158
6    2  2   1528.164866   1551.102616
7    3  1   1528.165664   1551.103414
8    3  2   1529.270778   1556.796078
9    1  4   1594.276525   1621.801825
10   4  2   1602.576938   1634.689789
11   3  0   1609.487338   1627.837538
12   4  0   1610.223137   1633.160887
13   4  1   1611.723967   1639.249267
14   0  3   1658.731030   1677.081230
15   0  4   1658.815935   1681.753685
16   1  3   1659.882416   1682.820166
17   2  0   1680.120665   1693.883315
18   1  2   1707.772491   1726.122691
19   0  2   1756.145317   1769.907968
20   1  1   1776.533722   1790.296372
21   1  0   1870.432865   1879.607965
22   0  1   1932.012001   1941.187102
23   0  0   2367.846163   2372.433713
Models sorted by BIC

     p  q           aic           bic
0    4  4    921.814711    963.102661
1    2  4   1297.058431   1329.171281
2    3  4   1300.818700   1337.519100
3    2  1   1526.197900   1544.548100
4    2  2   1528.164866   1551.102616
5    3  1   1528.165664   1551.103414
6    2  3   1525.544582   1553.069882
7    3  2   1529.270778   1556.796078
8    4  3   1526.885758   1563.586158
9    1  4   1594.276525   1621.801825
10   3  0   1609.487338   1627.837538
11   4  0   1610.223137   1633.160887
12   4  2   1602.576938   1634.689789
13   4  1   1611.723967   1639.249267
14   0  3   1658.731030   1677.081230
15   0  4   1658.815935   1681.753685
16   1  3   1659.882416   1682.820166
17   2  0   1680.120665   1693.883315
18   1  2   1707.772491   1726.122691
19   0  2   1756.145317   1769.907968
20   1  1   1776.533722   1790.296372
21   1  0   1870.432865   1879.607965
22   0  1   1932.012001   1941.187102
23   0  0   2367.846163   2372.433713
```

The best model order is (4,4). Let's use this order to fit the data and then diagnose the model by analyzing the model residuals.

```
# Create and fit model
model = sm.tsa.statespace.SARIMAX(co2, order=(4,1,4), trend='c')
results = model.fit()

# Create the 4 diagnostic plots
results.plot_diagnostics(figsize=(10,10))
plt.show()

# Print summary
print(results.summary())
```

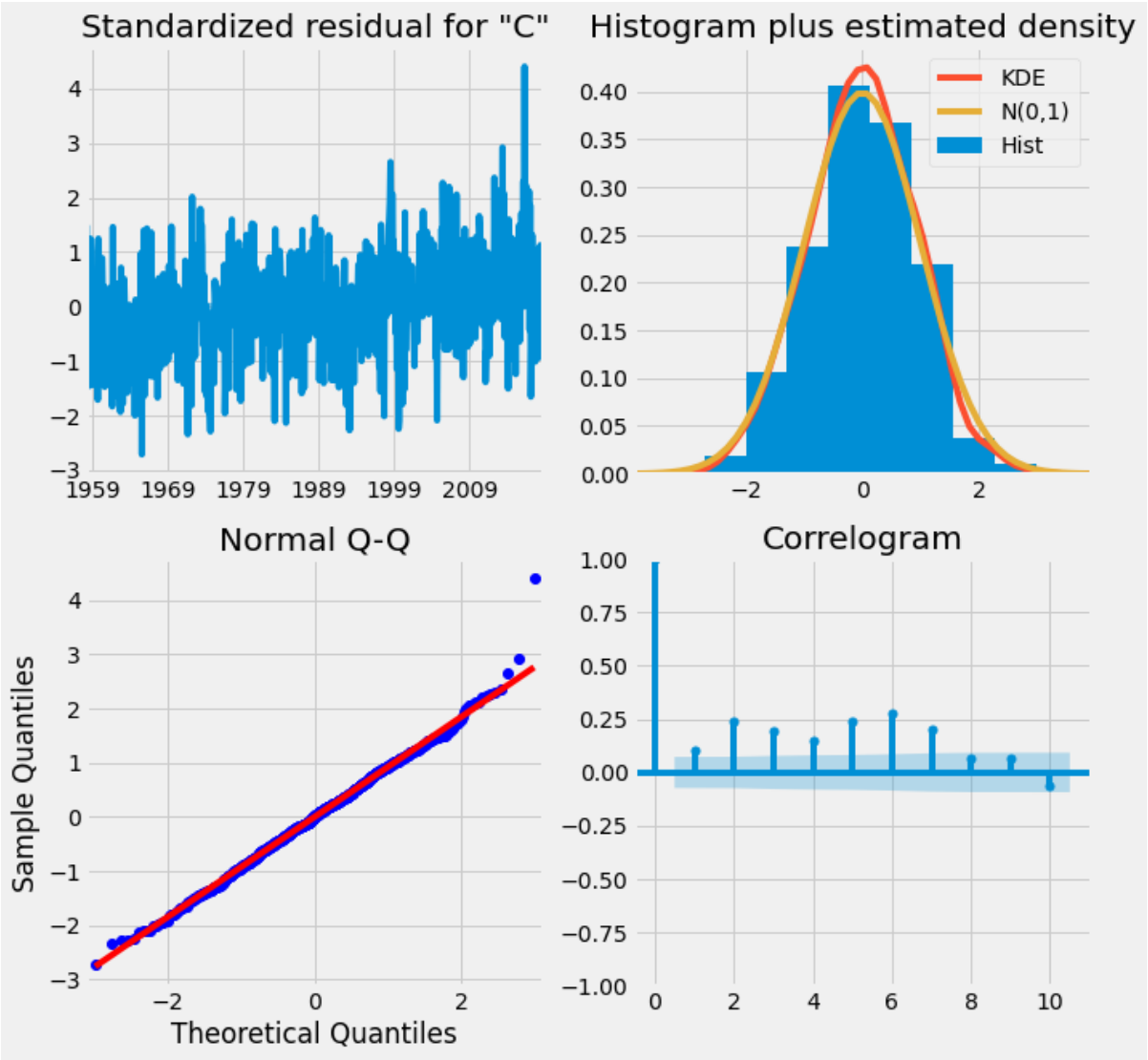


Figure 4.15. The diagnostics plots for the best-fitting model.

From the plots, we can make a decision that the model is doing well on the data, and no need to do another iteration.

4. Seasonal ARIMA Models

In the final section, we will discuss how to use seasonal **ARIMA** models to fit more complex data. You'll learn how to decompose this data into seasonal and non-seasonal parts, and then you'll get the chance to utilize all your **ARIMA** tools on one last global forecast challenge.

4.1. Introduction to seasonal time series

A **seasonal time series** has predictable patterns that repeat regularly. Although we call this feature seasonality, it can repeat after any length of time. These seasonal cycles might repeat every year, like sales of suncream, every week, like the number of visitors to a park, or every day, like the number of users on a website at any hour.

Any time series can be thought of to be made of three main components: trend, seasonal component, and the residual. The figure below shows the three components of the candy production time series used before.

```
from statsmodels.tsa.seasonal import seasonal_decompose

# Decompose data
decomp_results = seasonal_decompose(candy, period=12)

# Plot decomposed data
plt.rcParams["figure.figsize"] = (10,15)
decomp_results.plot()
plt.show()
```

From **Figure 4.16**, it is obvious that there is a seasonality in the data. The cycle here is 12 months. However, this was given as a parameter to the function. If we would like to find the cycle or the seasonality cycle, one method is to use the ACF and observe the lag after which the correlation pattern gets repeated. However, it is important to make sure that the data is stationary.

Since the candy production data is not stationary as it contains trends, we can transform it into a stationary series by taking the first difference. However, this time we are only trying to find the period of the time series, and the ACF plot will be clearer if we just subtract the rolling mean.

We calculate the rolling mean using the DataFrame's **.rolling()** method, where we pass the window size and also use the **.mean()** method. Any large window size, N, will work for this. We subtract this from the original time series and drop the NA values. The time series is now stationary, and it will be easier to interpret the ACF plot shown in Figure 4.17.

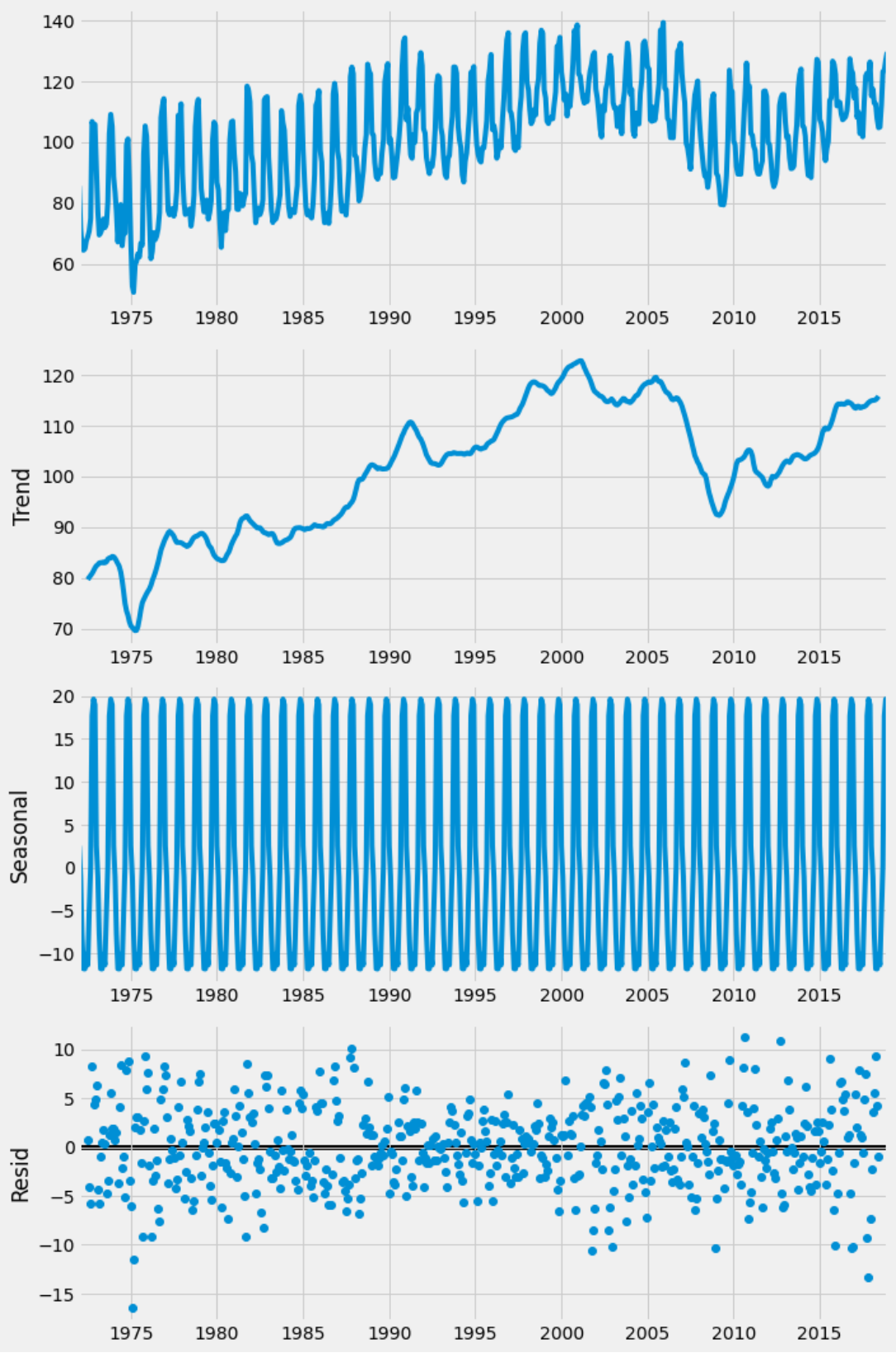


Figure 4.16. The candy production time series and its three components.

```
# Subtract long rolling average over 5 steps
candy = candy - candy.rolling(5).mean()

# Drop NaN values
candy = candy.dropna()

# Identifying seasonal data using ACF

# Create figure
fig, ax = plt.subplots(1,1, figsize=(8,4))

# Plot ACF
plot_acf(candy.dropna(), ax=ax, lags=25, zero=False)
plt.show()
```

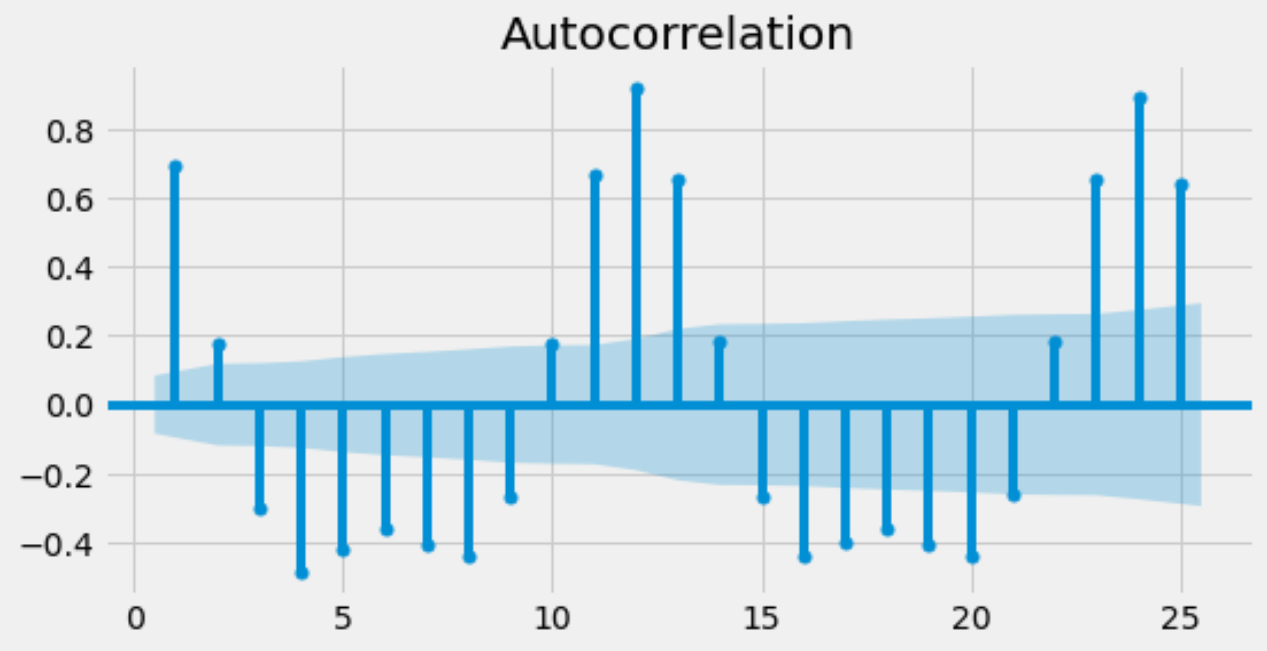


Figure 4.17. Autocorrelation of the detrended data.

After plotting the ACF of the detrended data, we can clearly see that there is a seasonal period of 12 steps. Since the data is seasonal, we will always have correlated residuals left if we try to fit an ARIMA model to it.

This means that we aren't using all of the information in the data, and so we aren't making the best predictions possible. This will be solved by using the seasonal Arima, as will be shown in the next subsection.

4.2. Seasonal ARIMA model

In the previous subsection, we discussed how to find seasonality in the data; in this subsection, we will discuss how to use the seasonality to make more accurate predictions. The seasonal

ARIMA (SARIMA) is used for this. Fitting a SARIMA model is like fitting two different ARIMA models at once, one to the seasonal part and another to the non-seasonal part.

Since we have these two models, we will have two sets of orders. We have non-seasonal orders for the autoregressive (p), the difference (d), and the moving average (q) parts. We also have this set of orders for the non-seasonal part added a new order, S, which is the length of the seasonal cycle.

Let's compare the SARIMA and ARIMA models further. Below is the equation of the ARIMA model with parameters (2,0,1). In this equation, we regress the time series against itself at lags 1 and 2 and against the shock at lag 1.

- **ARIMA(2,0,1) model:** $\boldsymbol{y}(t) = a(1)y(t-1) + a(2)y(t-2) + m(1) \in (t-1) + \in (t)$

Below is the equation for a simple SARIMA model with a season length of 7 days. This SARIMA model only has a seasonal part; we have set the non-seasonal orders to zero. We regress the time series against itself at lags of one season and two seasons and against the shock at a lag of one season.

- **SARIMA(0,0,0)(2,0,1)7 model:** $y = a(7)y(t-7) + a(14)y(t-14) + m(7) \in (t-7) + \in (t)$

This particular **SARIMA** model will be able to capture seasonal, weekly patterns, but won't be able to capture local, day-to-day patterns. If we construct a SARIMA model and include non-seasonal orders as well, then we can capture both of these patterns.

Fitting a **SARIMA** model is almost the same as fitting an ARIMA model. We import the model object and fit it as before. The only difference is that we have to specify the seasonal order as well as the regular order when we instantiate the model.

```
model = SARIMAX(df, order=(p,d,q), seasonal_order=(P,D,Q,S))
```

This means that there are a lot of model orders we need to find. In the last subsection, we learned how to find the seasonal period, S, using the ACF. The next task is to find the order of seasonal differencing.

To make a time series stationary, we may need to apply **seasonal differencing**. In seasonal differencing, instead of subtracting the most recent time series value, we subtract the time series value from one cycle ago. We can take the seasonal difference by using the **df.diff()** method. This time, we pass in an integer S, the length of the seasonal cycle.

If the time series shows a trend, then we take the normal difference. Once we have found the two orders of differencing and made the time series stationary, we need to find the other model orders. To find the non-seasonal orders (p,q), we plot the ACF and the PACF of the differenced time series. To find the seasonal orders (P, Q), we plot the ACF and PACF of the differenced

time series at multiple seasonal steps. Then we can use the same table of ACF and PACF rules to work out the seasonal order.

We make these seasonal ACF and PACF plots using the **plot_acf** and **plot_pacf** functions, but this time we set the lags parameter to a list of lags instead of a maximum. This plots the ACF and PACF at these specific lags only.

Let's apply this to the candy production time series used in the previous subsection. We know that the seasonal period of the data is 12 steps. So first we will take the seasonal difference to remove the seasonal effect, and then the one-step difference to remove the trend effect.

```
# Seasonal differencing
S = 12
candy_diff = candy.diff(S)
candy_diff.plot()
```

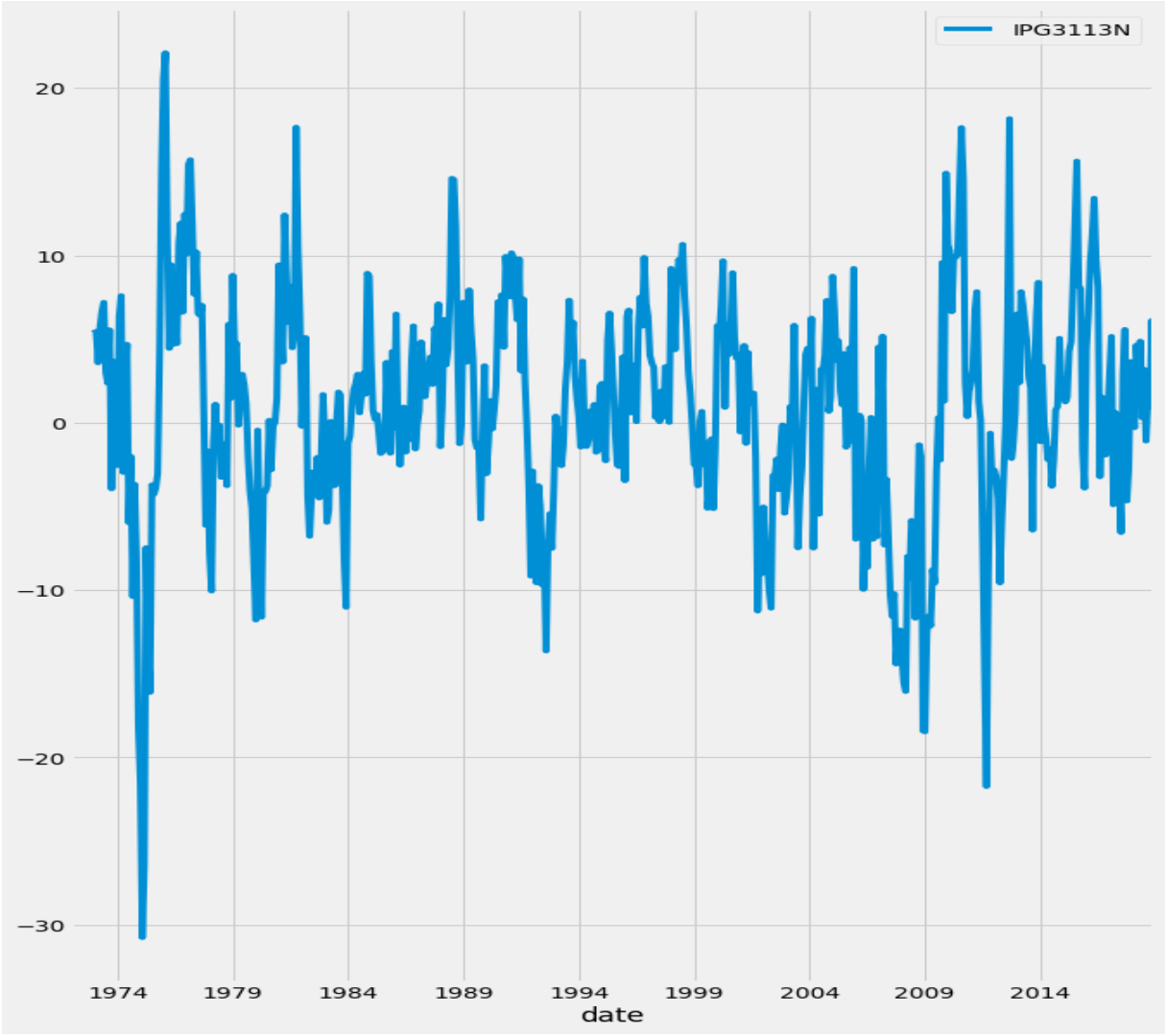


Figure 4.18. The candy production time series after removing the seasonal effect.

```
# one step differencing
candy_diff = candy.diff()
candy_diff = candy_diff.dropna()
candy_diff.plot()
```

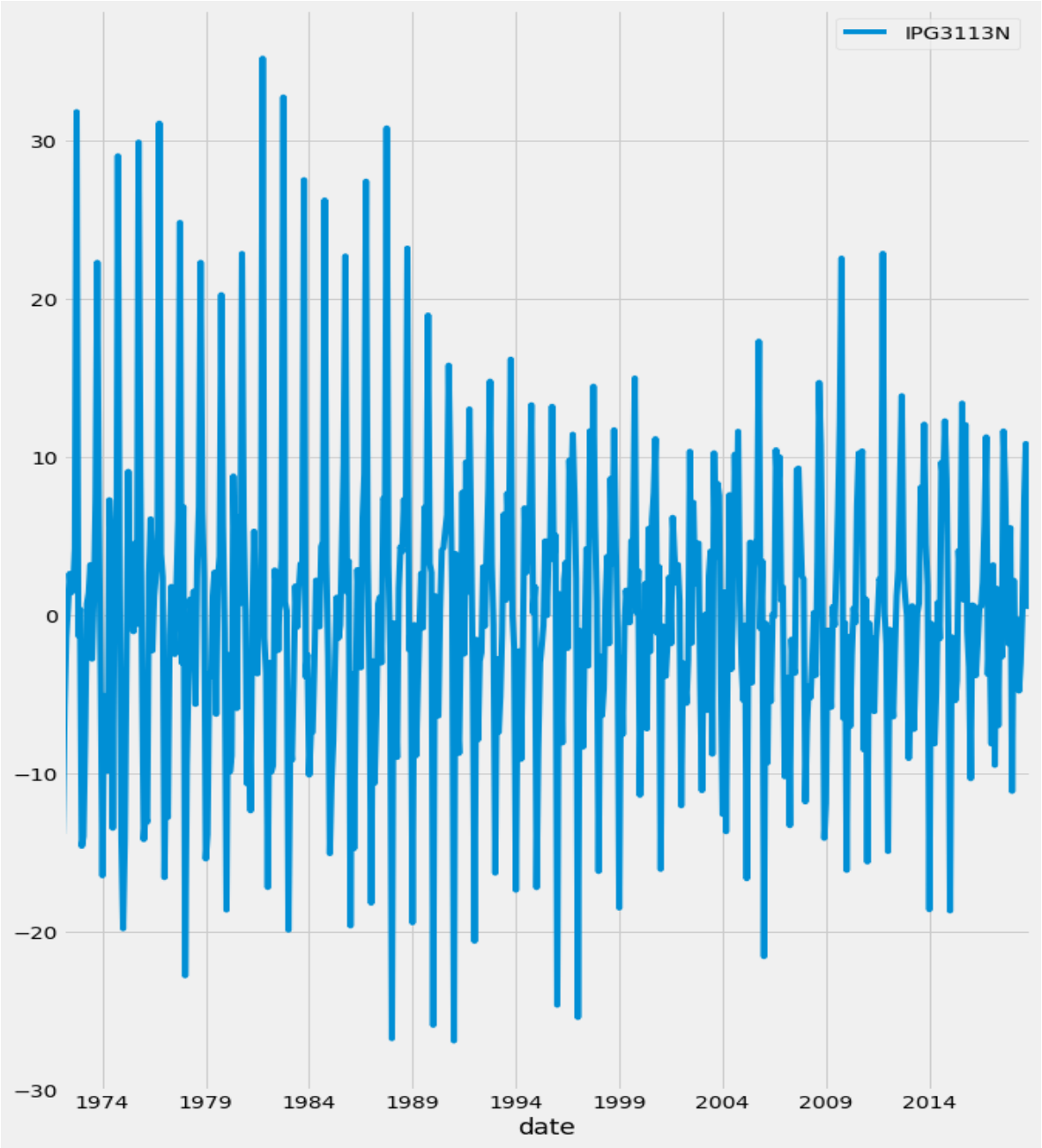


Figure 4.19. The candy production time series after removing the seasonal and trend effects.

Applying the augmented **Dickey-Fuller** test to make sure that the time series is now stationary.

```
# Run Dicky-Fuller test
result = adfuller(candy_diff)
```

```
# Print test statistic
print('The test stastics:', result[0])

# Print p-value
print("The p-value:",result[1])
```

```
The test stastics: -6.175912489755653
The p-value: 6.631549159335702e-08
```

The p-value is less than 0.05, so we can reject the null hypothesis and assume that the time series is now stationary. After that, we will plot the ACF and PACF to find the non-seasonal model parameters in Figure 4.20.

```
# find the non-seasonal model parameters
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf

# Create figure
fig, (ax1, ax2) = plt.subplots(2,1, figsize=(12,8))

# Plot the ACF of savings on ax1
plot_acf(candy_diff, lags=20, zero=False, ax=ax1)

# Plot the PACF of savings on ax2
plot_pacf(candy_diff, lags=20, zero=False, ax=ax2)

plt.show()
```

There is no clear pattern in the ACF and PACF plots, so we will use the AIC and BIC to narrow down our choices.

```
import statsmodels.api as sm
order_aic_bic =[]

# Loop over p values from 0-4
for p in range(5):
    # Loop over q values from 0-4
    for q in range(5):

        try:
            # create and fit ARMA(p,q) model
            model = sm.tsa.statespace.SARIMAX(candy_diff, order=(p, 1,
q))
            results = model.fit()

            # Print order and results
            order_aic_bic.append((p, q, results.aic, results.bic))
        except:
            print(p, q, None, None)
```

```
# Make DataFrame of model order and AIC/BIC scores
order_df = pd.DataFrame(order_aic_bic, columns=['p', 'q',
'aic','bic'])

# lets sort them by AIC and BIC

# Sort by AIC
print("Models sorted by AIC ")
print("\n")
print(order_df.sort_values('aic').reset_index(drop=True))

# Sort by BIC
print("Models sorted by BIC ")
print("\n")
print(order_df.sort_values('bic').reset_index(drop=True))
```

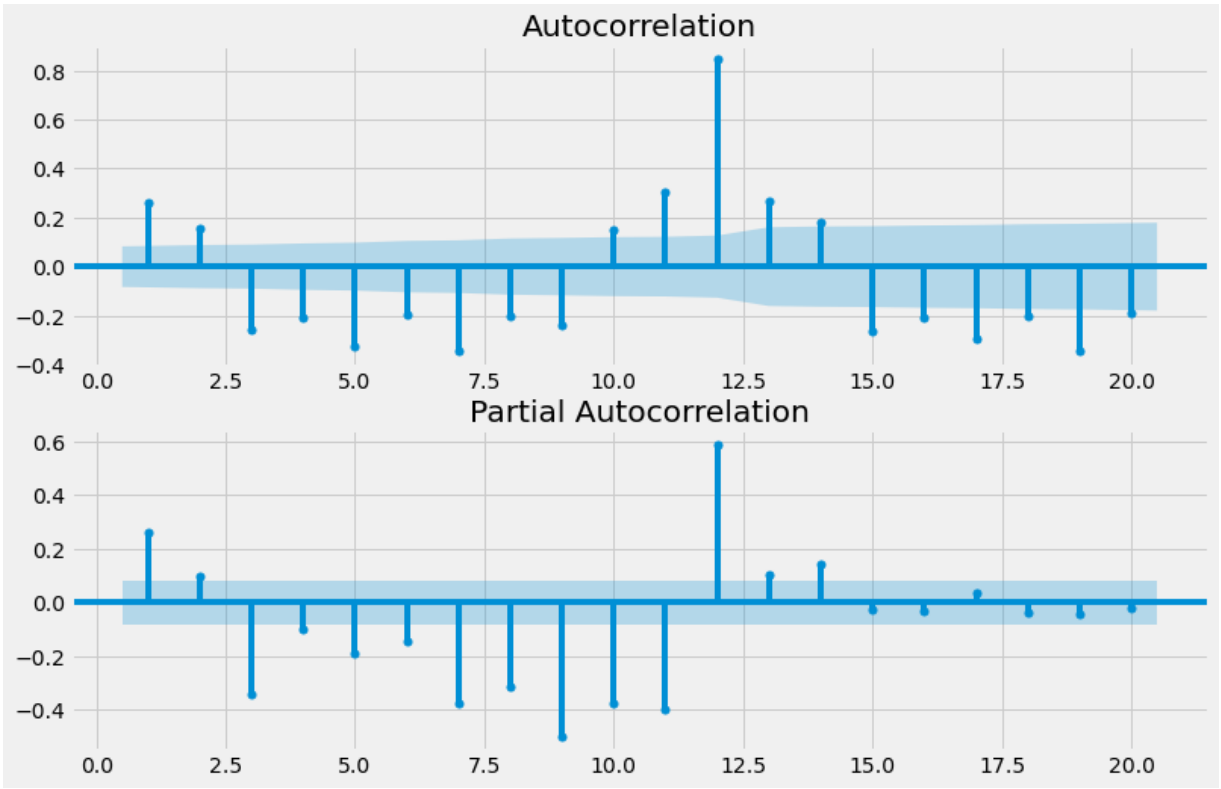


Figure 4.20. The ACF and PACF of the candy production time series after applying the seasonal and one-step difference.

The best model parameters are (0,3). After that, we will plot the seasonal **ACF** and **PACF** to find the seasonal parameters.

```
# Plotting seasonal ACF and PACF

# Create figure
fig, (ax1, ax2) = plt.subplots(2,1)
```

```
# Plot seasonal ACF
plot_acf(candy_diff, lags=[12,24,36,48,60,72, 84, 96 ], ax=ax1)

# Plot seasonal PACF
plot_pacf(candy_diff, lags=[12,24,36,48,60,72, 84, 96], ax=ax2)
plt.show()
```

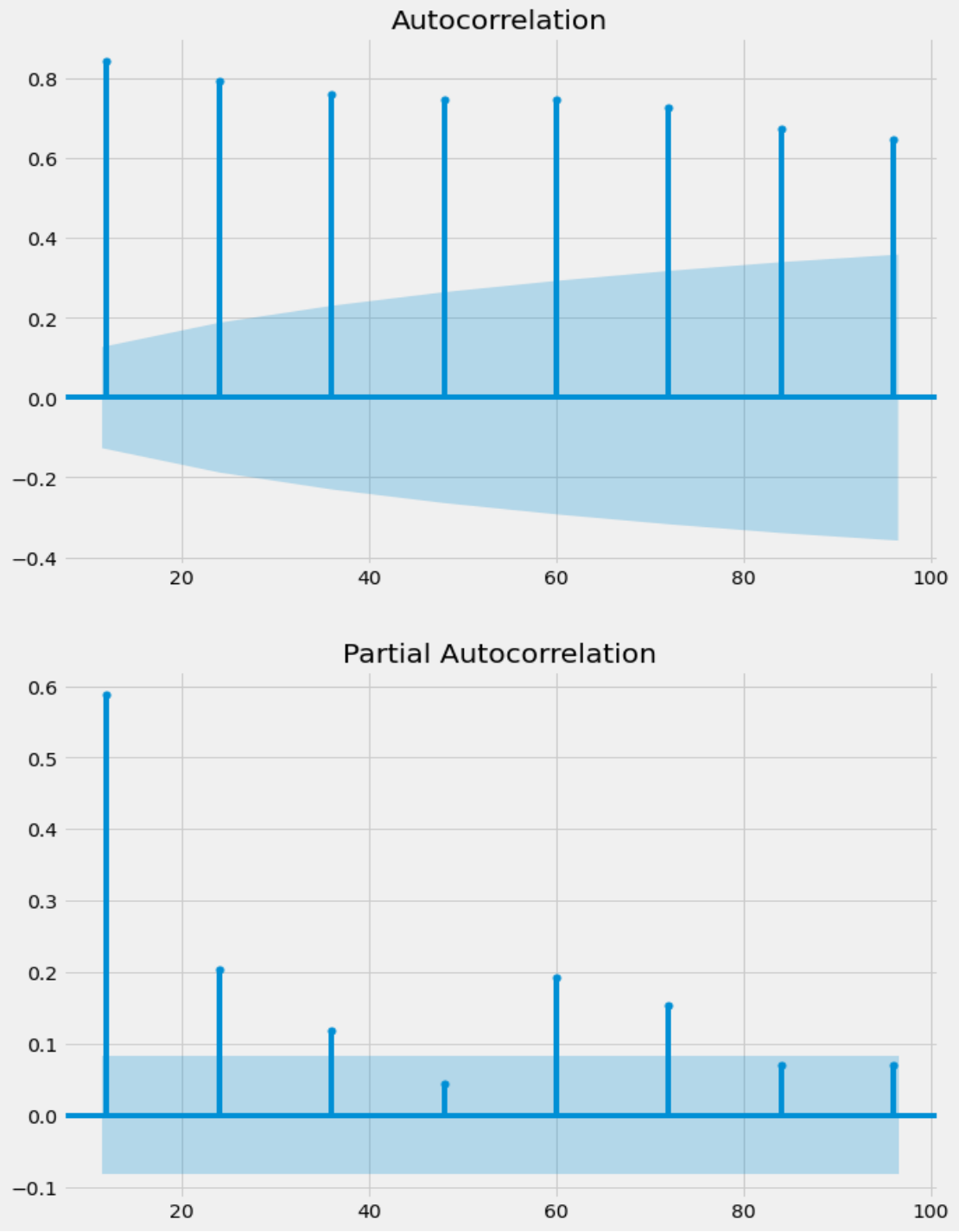


Figure 4.21. The seasonal ACF and PACF of the candy production time series.

The **ACF** is tailing off, and the **PACF** is cut off after a lag of three. So the model parameter is (0,3). The final step is to fit the model with all of these parameters.

```
from statsmodels.tsa.statespace.sarimax import SARIMAX

# Instantiate model
S = 12
D = 1
d = 1
P = 0
Q = 3
p = 4
q = 3
model = SARIMAX(candy, order=(p,d,q), seasonal_order=(P,D,Q,S))

# Fit model
results = model.fit()
```

4.3. Process automation and model saving

Previously, we searched for the ARIMA model order using for loops. Now that we have seasonal orders as well, this is very complex. Fortunately, there is a package that will do most of this work for us. This is the **pmdarima** package.

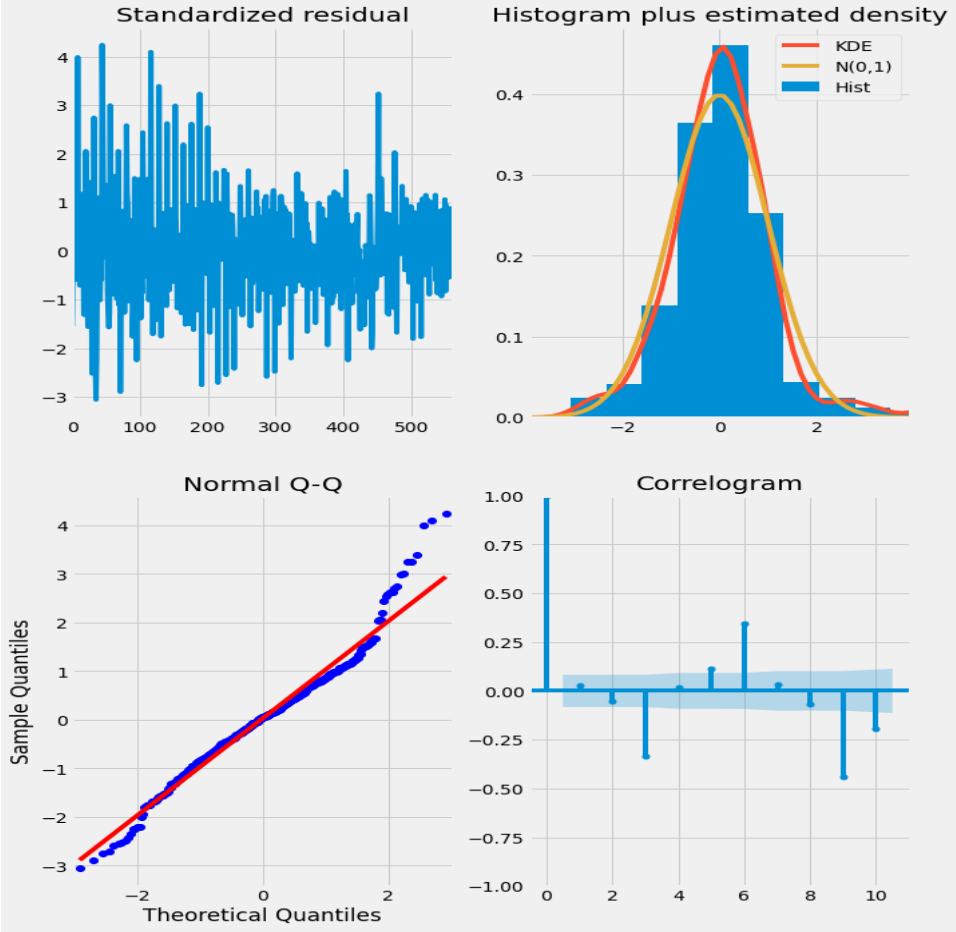
The **auto_arima** function from this package loops over model orders to find the best one. The object returned by the function is the results object of the best model found by the search. This object is almost exactly like a statsmodels **SARIMAX** results object and has the summary and the plot-diagnostics method shown in Figure 4.22.

```
# Searching over model orders
import pmdarima as pm

results = pm.auto_arima(candy)
print(results)
print(results.summary())
results.plot_diagnostics()
```

The order of the best model found by the search is (2,3). However, it did not consider the seasonal components. To do this, we have to set the seasonal parameter to true. We also need to specify the length of the seasonal period and the order of seasonal differencing. Finally, we will use a few non-order parameters. Using **information_criterion,** you can select whether to choose the best model based on AIC or BIC. If **information_criterion** =’aic’, it will select it based on AIC, and if it is information_criterion ‘bic’, it will select it based on BIC.

Also, we will set the **trace parameter** to true, then this function prints the AIC and BIC for each model it fits. To ignore bad models, we will set the **error_action = ‘ignore’**. We will set the **stepwise** to true, then instead of searching over all of the model orders, the function searches outward from the initial model order guess using an intelligent search method. This might save running time and computation power.



ARIMA(2,1,3)(0,0,0)[0]

SARIMAX Results

Dep. Variable:	y	No. Observations:	564
Model:	SARIMAX(2, 1, 3)	Log Likelihood	-1860.785
Date:	Tue, 03 May 2022	AIC	3733.570
Time:	12:17:23	BIC	3759.570
Sample:	0	HQIC	3743.720
	- 564		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
ar.L1	1.7056	0.011	160.814	0.000	1.685	1.726
ar.L2	-0.9692	0.012	-83.805	0.000	-0.992	-0.947
ma.L1	-1.9864	0.041	-48.177	0.000	-2.067	-1.906
ma.L2	1.2847	0.079	16.276	0.000	1.130	1.439
ma.L3	-0.2043	0.044	-4.687	0.000	-0.290	-0.119
sigma2	43.1480	2.242	19.248	0.000	38.754	47.542

Ljung-Box (L1) (Q):	0.44	Jarque-Bera (JB):	107.29
Prob(Q):	0.51	Prob(JB):	0.00
Heteroskedasticity (H):	0.42	Skew:	0.36
Prob(H) (two-sided):	0.00	Kurtosis:	5.01

Figure 4.22. The summary statistics and the parameters of the best SARIMAX model.

```
# Seasonal search parameters

results = pm.auto_arima(candy, # data
                        seasonal=True, # is the time series seasonal
                        m=12, # the seasonal period
                        D=1, # seasonal difference order
                        start_P=1, # initial guess for P
                        start_Q=1, # initial guess for Q
                        max_P=4, # max value of P to test
                        max_Q=4, # max value of Q to test
                        information_criterion='aic', # used to select
best model
                        trace=True, # print results whilst training
                        error_action='ignore', # ignore orders that
don't work
                        stepwise=True,
                       )
print(results)
print(results.summary())
results.plot_diagnostics()
```

Here are the parameters of the best model:

```
Best model:  ARIMA(4,0,3)(0,1,1)[12]
Total fit time: 829.962 seconds
 ARIMA(4,0,3)(0,1,1)[12]
```

Here are the summary statistics and the parameters of the best SARIMA model.

SARIMAX Results

Dep. Variable:	y	No. Observations:	564
Model:	SARIMAX(4, 0, 3)x(0, 1, [1], 12)	Log Likelihood	-1494.635
Date:	Tue, 03 May 2022	AIC	3007.270
Time:	13:21:23	BIC	3046.092
Sample:	0	HQIC	3022.438
	- 564		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
ar.L1	0.4041	0.104	3.880	0.000	0.200	0.608
ar.L2	0.2342	0.026	9.083	0.000	0.184	0.285
ar.L3	0.8856	0.018	50.112	0.000	0.851	0.920
ar.L4	-0.5590	0.097	-5.791	0.000	-0.748	-0.370
ma.L1	0.3320	0.082	4.057	0.000	0.172	0.492
ma.L2	0.0358	0.091	0.392	0.695	-0.143	0.215
ma.L3	-0.7741	0.077	-10.012	0.000	-0.926	-0.623
ma.S.L12	-0.7054	0.032	-22.043	0.000	-0.768	-0.643
sigma2	12.9833	0.687	18.889	0.000	11.636	14.330

Ljung-Box (L1) (Q):	0.28	Jarque-Bera (JB):	41.67
Prob(Q):	0.60	Prob(JB):	0.00
Heteroskedasticity (H):	1.21	Skew:	-0.38
Prob(H) (two-sided):	0.19	Kurtosis:	4.11

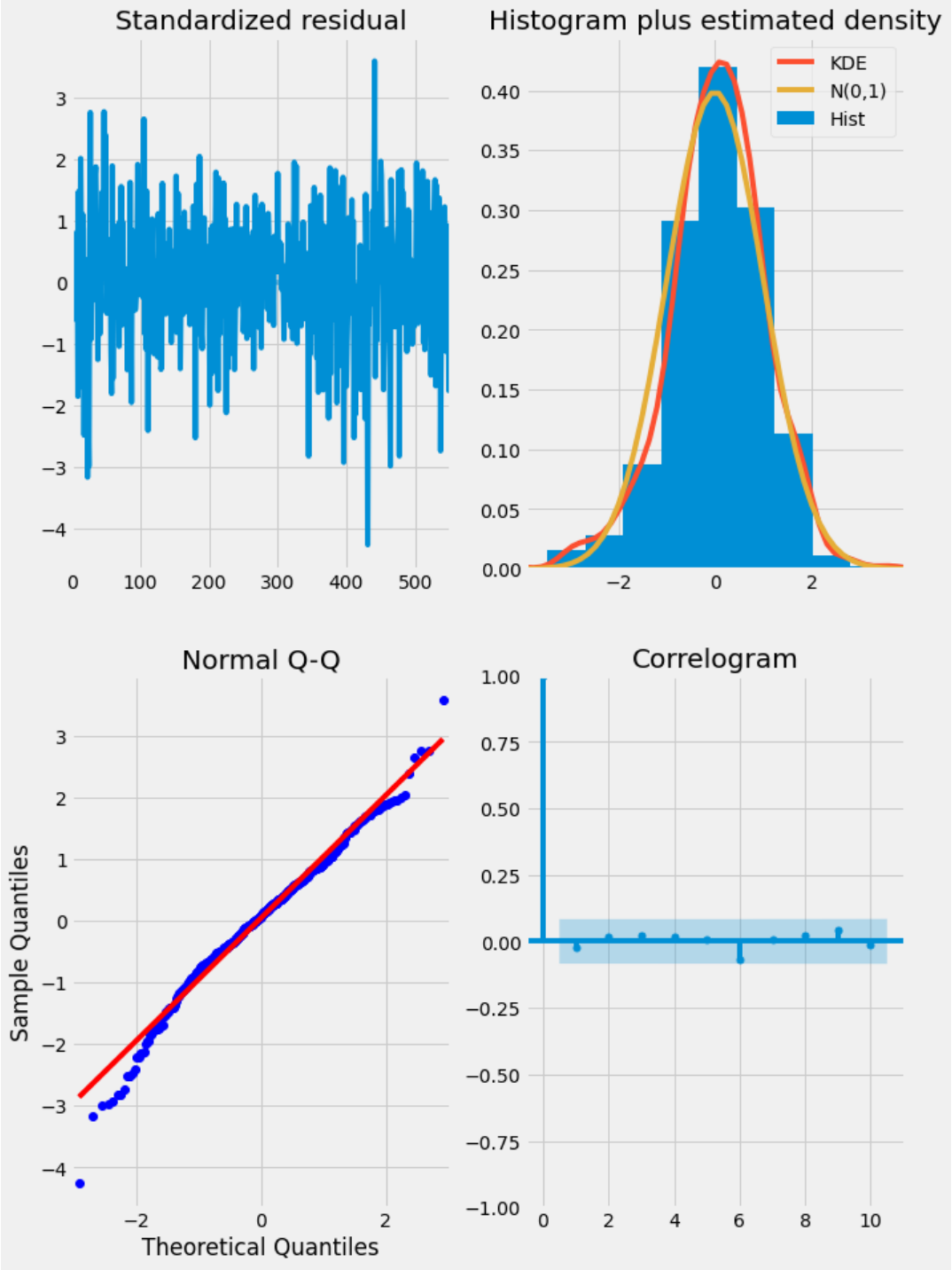


Figure 4.23. The residual diagnostics of the best SARIMA model.

Once you have fit a model in this way, you may want to save it and load it later. You can do this using the **joblib** package. To save the model, we use the dump function from the **joblib** package. We pass the model results object and the **file_path** into this function. Later on, when we want to make new predictions, we can load this model again. To do this, we use the load function from **joblib**.

```
import joblib

# Select a filepath
filepath = 'model.pkl'

# Save model to filepath
joblib.dump(results, filepath)

# Load the model
filepath ='model.pkl'

# Load model object from filepath
loaded_model = joblib.load(filepath)
```

If you have new data and you would like to update the trained model, you can use the **model.update(new_data).** This isn't the same as choosing the model order again, and so if you are updating with a large amount of new data, it may be best to go back to the start of the Box-Jenkins method.

2.4. SARIMA and Box-Jenkins for seasonal time series

We previously covered the Box-Jenkins method for ARIMA models. We go through the identification of the model order, estimating or fitting the model, diagnosing the model residuals, and finally, production.

For SARIMA models, the only step in the method that will change is the identification step. At the identification step, we add the tasks of determining whether a time series is seasonal, and if so, then finding its seasonal period.

We also need to consider transforms to make the seasonal time series stationary, such as seasonal and non-seasonal differencing and other transforms. Sometimes we will have the choice of whether to apply seasonal differencing, non-seasonal differencing, or both to make a time series stationary.

Some good rules of thumb are that you should never use more than one order of seasonal differencing, and never more than two orders of differencing in total. Sometimes you will be able to make a time series stationary by using either one seasonal difference or one non-seasonal difference.

There are two types of seasonality: weak seasonality and strong seasonality. In weak seasonality, the seasonal oscillations in the time series will not always look the same and will be harder to identify.

While in the strong seasonality, the seasonality pattern in the time series will be strong. When you have a strong seasonal pattern, you should always use one order of seasonal differencing. This will ensure that the seasonal oscillation will remain in your dynamic predictions far into the future without fading out. While in a weak seasonal pattern, use it if necessary.

Just like in ARIMA modeling, sometimes we need to use other transformations on our time series before fitting. If the seasonality is **Additive seasonality** in which the seasonal pattern just adds or takes away a little from the trend.

Whenever the seasonality is **additive,** we shouldn't need to apply any transforms except differencing. If the seasonality is **multiplicative,** the amplitude of the seasonal oscillations will get larger as the data trends up or smaller as it trends down. If the seasonality is **multiplicative**, the SARIMA model can't fit this without extra transforms. To deal with this, we take the log transform of the data before modeling it.

Conclusion

In this chapter, we have transformed the abstract concept of "predicting the future" into a concrete, mathematical process. We moved beyond simple trend lines and learned to model the internal structure of time series data.

You have mastered the essential workflow of classical forecasting. You learned that effective modeling begins with stationarity, using differencing to stabilize trends and variance. You combined autoregression and moving averages to create ARIMA models that learn from past values and past errors. Crucially, you learned that a model is only as good as its diagnostics—using residual analysis to ensure that no patterns are left unexplained.

Finally, we tackled the complexity of the real world by introducing SARIMA, allowing us to capture repeating seasonal cycles, and leveraged the pmdarima library to automate the often-tedious search for optimal hyperparameters. You now possess a complete, end-to-end framework—the Box-Jenkins method—that allows you to take a raw dataset, identify its structure, and generate reliable forecasts with confidence intervals. These statistical foundations are critical as we look toward even more advanced machine learning techniques in the future

Time Series Forecasting with Machine Learning

Table of Contents:

This final chapter of the Hands-On Time Series Analysis with Python provides a practical project for applying machine learning techniques to time series forecasting. Beginning with an overview of time series types, real-world applications, and the intersection of machine learning with temporal data, the article then guides readers through building and refining predictive models.

Key topics include generating time-aware features, implementing advanced forecasting strategies, and understanding temporal dependencies. The course also covers best practices for model validation, including cross-validation tailored for time-dependent data, as well as methods to assess stationarity and stability.

By the end, you will have the skills to develop, evaluate, and improve machine learning models for real-world time series problems using Python.

1. Introduction to Time Series and Machine Learning

Time series data is ubiquitous. Whether it be stock market fluctuations, sensor data recording, climate change, or activity in the brain, any signal that changes over time can be described as a time series. Machine learning has emerged as a powerful method for leveraging complexity in data in order to generate predictions and insights into the problem one is trying to solve.

This chapter is an intersection between these two worlds of machine learning and time-series data and covers feature engineering and other advanced techniques in order to efficiently predict stock prices. The first section will be an introduction to the basics of machine learning, time-series data, and the intersection between them.

1.1. Time series kinds and applications

A time-series data is data that changes over time. This can take many different forms, such as atmospheric CO2 over time, the waveform of your voice, the fluctuation of a stock's value over the year, or demographic information about a city.

Time series data consists of at least two things: One, an array of numbers that represents the data itself. Two, another array that contains a timestamp for each datapoint. The timestamps can include a wide range of time data, from months of the year to nanoseconds.

Machine learning has taken the world of data science by storm. In the last few decades, advances in computing power, algorithms, and community practices have made it possible to use computers to ask questions that were never thought possible.

Machine learning is about finding patterns in data—often patterns that are not immediately obvious to the human eye. This is often because the data is either too large or too complex to be processed by a human.
Another crucial part of machine learning is that we can build a model of the world that formalizes our knowledge of the problem at hand. We can use this model to make predictions. Combined with automation, this can be a critical component of an organization's decision-making.

Since machine learning is all about finding patterns in data and since time-series data always changes over time, this turns out to be a useful pattern to utilize. In this article, we will focus on a simple machine learning pipeline in the context of time-series data.

This boils down to the following main steps. Feature extraction: What kinds of special features leverage a signal that changes over time? Model fitting: What kinds of models are suitable for asking questions with time-series data? Validation: How can we validate a model that uses time-series data? What considerations must we make because it changes over time?

1.2. Machine learning and time-series data

In this subsection, we'll discuss the interaction between machine learning and time-series data and introduce why they're worth thinking about in tandem. First, let's give a quick overview of the data we'll be using. They're both freely available online and come from the excellent website Kaggle.

We will explore data from the **New York Stock Exchange**, which has a sampling frequency on the order of one sample per day. Our goal is to predict the stock value of a company using historical data from the market.

As we are predicting a continuous output value, this is a regression problem. Let's load the data and take a look at the raw data. Each row is a sample for a given day and company. It seems that the dates go back all the way to 2010.

```
# Load the New York stock exchange prices
prices = pd.read_csv('prices.csv', index_col='date',
            parse_dates=True)

prices.head()
```

Here is the first five rows of the New York stock prices time series.

date	symbol	close	volume
2010-01-04	AAPL	214.009998	123432400.0
2010-01-04	ABT	54.459951	10829000.0
2010-01-04	AIG	29.889999	7750900.0
2010-01-04	AMAT	14.300000	18615100.0
2010-01-04	ARNC	16.650013	11512100.0

Let's plot the change in the prices of the stock in the FiveThirtyEight style using **matplotlib**

```
# change the plot style into fivethirtyeight
plt.style.use('fivethirtyeight')

# Plot and show the time series on axis ax1
fig, ax1 = plt.subplots()
prices['close'].plot(ax=ax1, figsize=(12,10))
plt.title('New York stock prices change')
plt.xlabel('Date')
plt.ylabel('Stock prices')
plt.show()
```

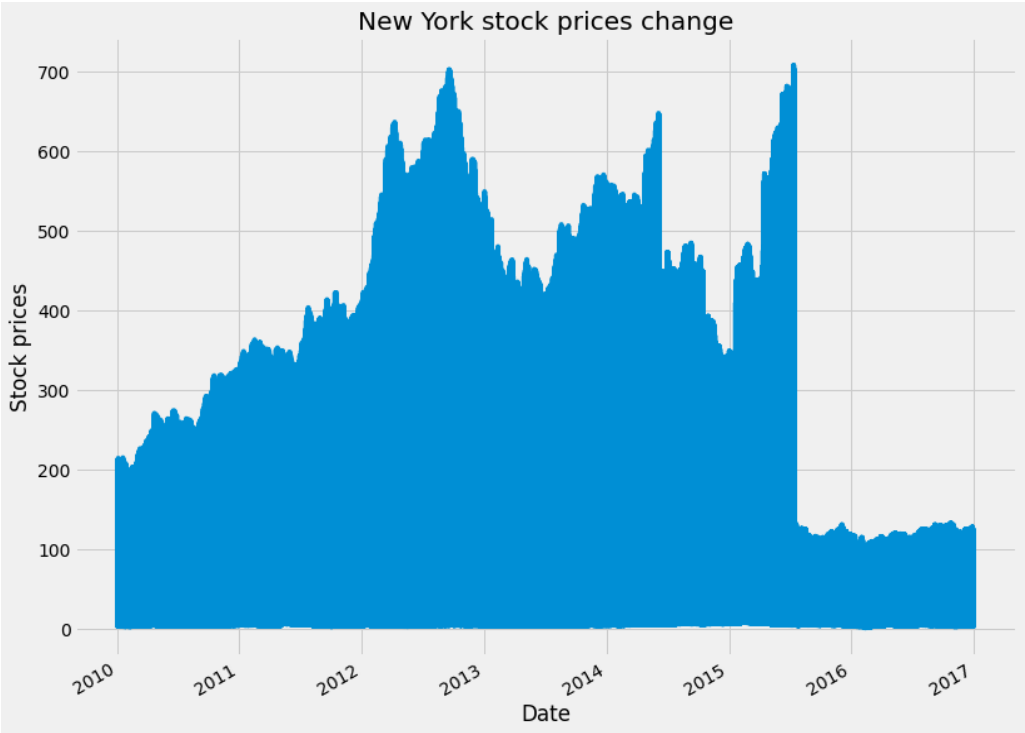


Figure 5.1. New York stock prices change with time.

It is also useful to investigate the "type" of data in each column. Numpy or Pandas may treat an array of data in special ways depending on its type. We can print the type of each column by looking at the **.dtypes()** attribute.

Here we see that the type of symbol column is "object" and the close and volume columns are **float 64**, which is a generic data type. Also, we will convert the index format into a date datetime for better indexing using **.to_datetime()**

```
# convert the type of the index from object to time
prices.index = pd.to_datetime(prices.index)
```

2. Time Series Forecasting with Machine Learning

If you want to predict patterns from data over time, there are special considerations to take in how you choose and construct your model. This section covers how to gain insights into the data before fitting your model, as well as best practices in using predictive modeling for time series data.

2.1. Predicting data over time

In this section, we will on regression, and in the next article of this series, we will shift our focus from regression to classification. **Regression** has several features and caveats that are unique to time-series data.

The biggest difference between **regression** and **classification** is that regression models predict continuous outputs, whereas classification models predict categorical outputs. In the context of time series, this means we can have finer-grained predictions over time.

We'll begin by visualizing and predicting time-series data. Then, we'll cover the basics of cleaning the data, and finally, we'll begin extracting features that we can use in our models. We will be using the prices dataset, which is the market stock price of various large companies.

The total dataset can be found ***here***; however, we will use this ***preprocessed subset*** of the stock prices dataset for the sake of simplicity and better visualization. Let's first upload the data and print the first five rows of it.

```
# Load the data
preprocessed_prices = pd.read_csv('preprocessed_prices.csv',
parse_dates=True, index_col='date')
```

date	AAPL	ABT	AIG	AMAT	ARNC	BAC	BSX	C	CHK	CMCSA	...	QCOM	RF	SBUX
2010-01-04	214.009998	54.459951	29.889999	14.300000	16.650013	15.690000	9.01	3.400000	28.090001	16.969999	...	46.939999	5.42	23.049999
2010-01-05	214.379993	54.019953	29.330000	14.190000	16.130013	16.200001	9.04	3.530000	28.970002	16.740000	...	48.070000	5.60	23.590000
2010-01-06	210.969995	54.319953	29.139999	14.160000	16.970013	16.389999	9.16	3.640000	28.650002	16.620001	...	47.599998	5.67	23.420000
2010-01-07	210.580000	54.769952	28.580000	14.010000	16.610014	16.930000	9.09	3.650000	28.720002	16.969999	...	48.980000	6.17	23.360001
2010-01-08	211.980005	55.049952	29.340000	14.550000	17.020014	16.780001	9.00	3.590000	28.910002	16.920000	...	49.470001	6.18	23.280001
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
2015-01-26	113.099998	44.150002	51.209999	24.240000	16.080012	15.850000	14.96	48.889999	20.139999	56.689999	...	72.610001	9.08	88.120003
2015-01-27	109.139999	43.680000	50.470001	23.570000	15.920012	15.630000	14.74	48.310001	20.250000	56.360001	...	71.769997	8.97	88.339996
2015-01-28	115.309998	43.410000	49.209999	23.379999	15.800012	15.200000	14.59	47.110001	19.129999	54.590000	...	70.989998	8.74	87.570000
2015-01-29	118.900002	45.259998	49.259998	23.620001	15.980012	15.430000	14.85	47.619999	19.209999	53.869999	...	63.689999	8.85	89.050003
2015-01-30	117.160004	44.759998	48.869999	22.840000	15.650013	15.150000	14.81	46.950001	19.180000	53.150002	...	62.459999	8.70	87.529999

1278 rows × 46 columns

We have the stock prices for **42 companies** changing over time, starting from 01/04/2010 to 01/30/2015, with the time stamp of one day. To get a better idea of how the data changes with time, we will plot the stock price change of Yahoo and eBay over time, as shown in Figure 5.2:

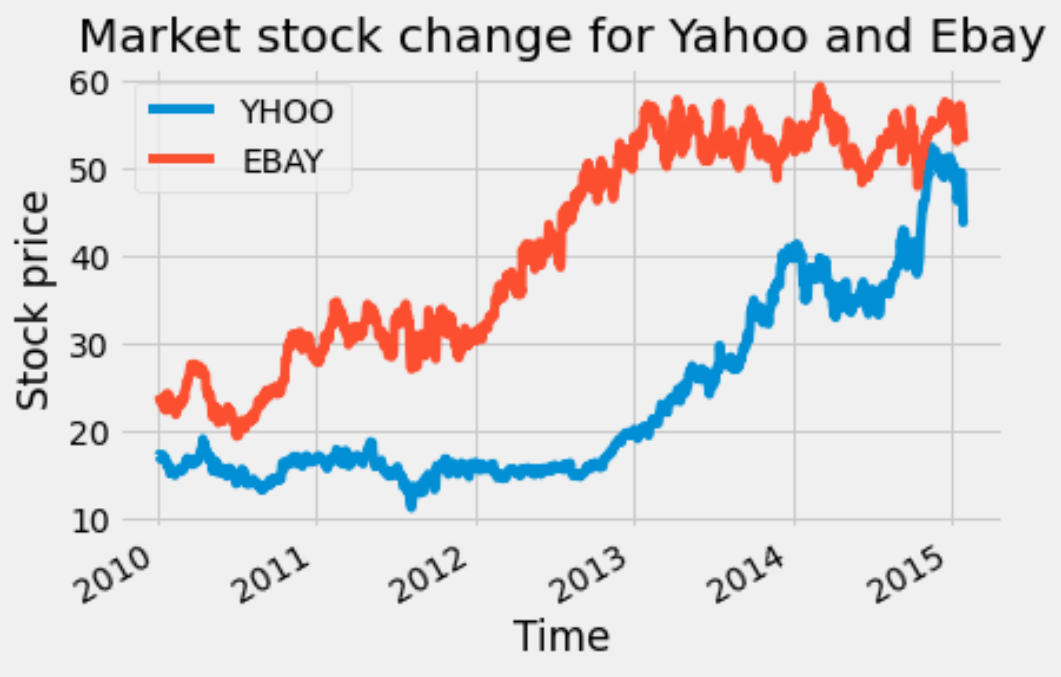


Figure 5.2. Market stock price change for Yahoo and eBay.

To understand the relationship between the stock price changes of the two companies, we can plot a scatter plot of both of them using the code below:

```
# Scatterplot with one company per axis
preprocessed_prices.plot.scatter('EBAY', 'YHOO')
plt.title('Scatter plot of Yahoo and Ebay')
plt.show()
```

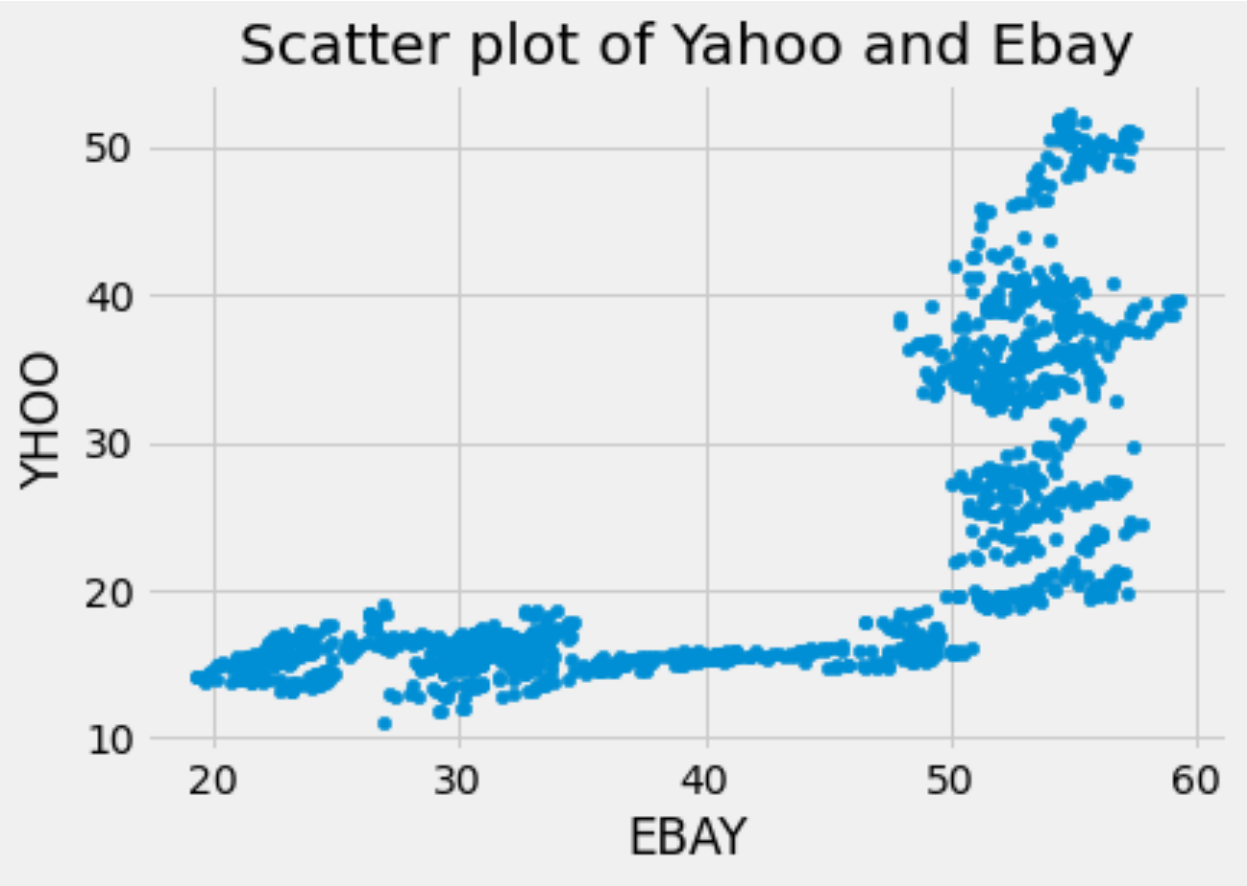


Figure 5.3. Scatter plot of Yahoo and eBay stock prices.

We can also code time as the color of each data point in order to visualize how the relationship between these two variables changes.

```
# Scatterplot with color relating to time
preprocessed_prices.plot.scatter('EBAY', 'YHOO',
c=preprocessed_prices.index,
                    cmap=plt.cm.viridis, colorbar=True)
plt.show()
```

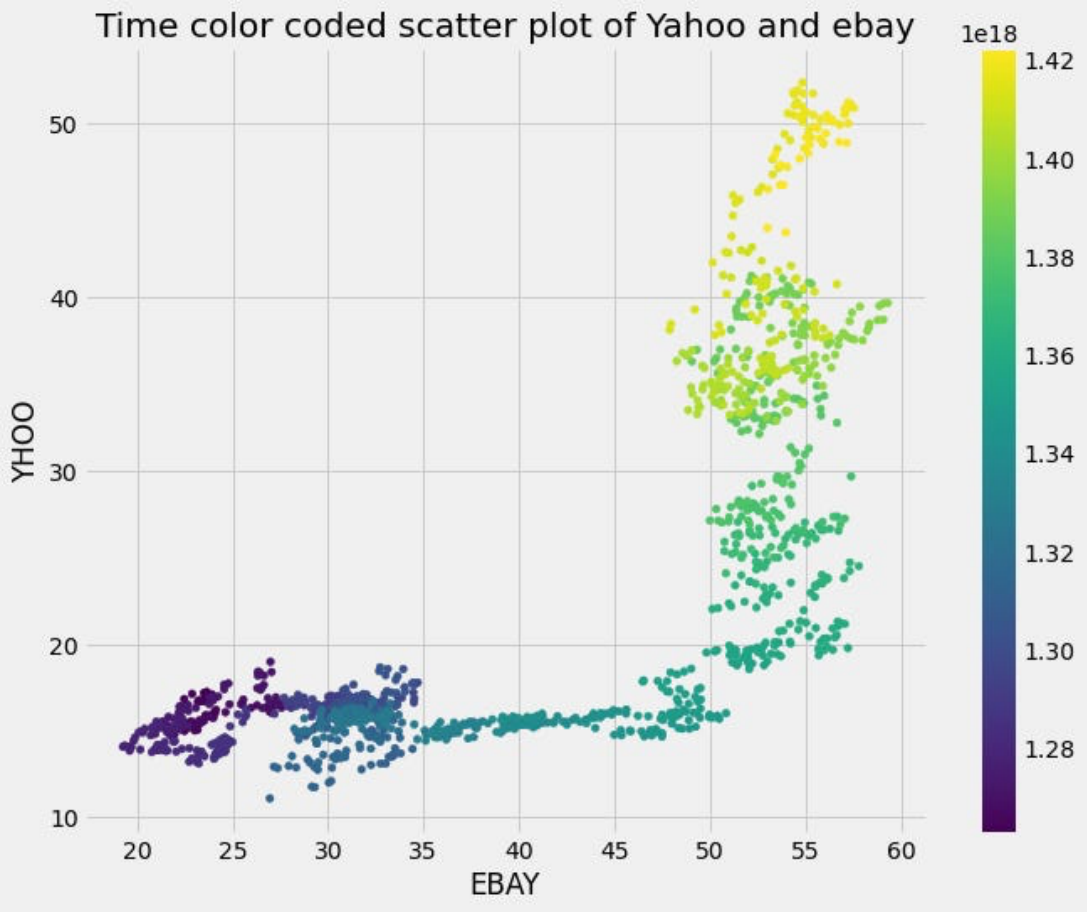


Figure 5.4. Time color-coded scatter plot of Yahoo and eBay stock prices.

Fitting regression models with **scikit-learn** is straightforward, due to the consistency in the API of **scikit-learn**, which is one of its greatest strengths. There is, however, a completely different subset of models that accomplish regression.

We'll begin by focusing on LinearRegression, which is the simplest form of regression. Here we see how you can instantiate the model, fit, and predict on training data. We will use the eBay, Nvidia, and Yahoo stock prices as the features, and the target value will be the Apple stock price. We will use the linear regression model for prediction from **scikit_learn.**

There are several options to evaluate the regression model. The simplest is the correlation coefficient, whereas the most common is the coefficient of determination, or R-squared, which we will use here.

The coefficient of determination can be summarized as the total amount of error in your model (the difference between predicted and actual values) divided by the total amount of error if you'd built a "dummy" model that simply predicted the output data's mean value at each time point. You subtract this ratio from "1", and the result is the coefficient of determination. It is bounded on top by "1", and can be infinitely low. We will evaluate the model using the r2_score function

from **scikit-learn**, which takes the predicted output values first and the "true" output values second.

```
# Fitting a simple regression model
from sklearn.linear_model import Ridge
from sklearn.model_selection import cross_val_score

# Use stock symbols to extract training data
X = preprocessed_prices[['EBAY', 'NVDA', 'YHOO']]
y = preprocessed_prices[['AAPL']]

# Fit and score the model with cross-validation
scores = cross_val_score(Ridge(), X, y, cv=3)
print(scores)
```

We can plot the predicted Apple stock price with the true values on the same plot to have a better understanding of the model output:

```
# Visualizing predicted values
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score

# Split our data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y,
train_size=.8, shuffle=False, random_state=1)

# Fit our model and generate predictions
model = Ridge()
model.fit(X_train, y_train)
predictions = model.predict(X_test)
score = r2_score(y_test, predictions)
print(score)

y_test['predictions'] = predictions
y_test.rename(columns= {'AAPL':'True_vlaue'}, inplace=True)

# Visualize our predictions along with the "true" values, and print
the score
fig, ax = plt.subplots(figsize=(15, 5))
ax.plot(y_test['True_vlaue'], color='k', lw=3)
ax.plot(y_test['predictions'], color='r', lw=2)
ax.legend(['True values', 'Predicitions'])
plt.title('Apple stock price true value and predicted price')
plt.show()
```

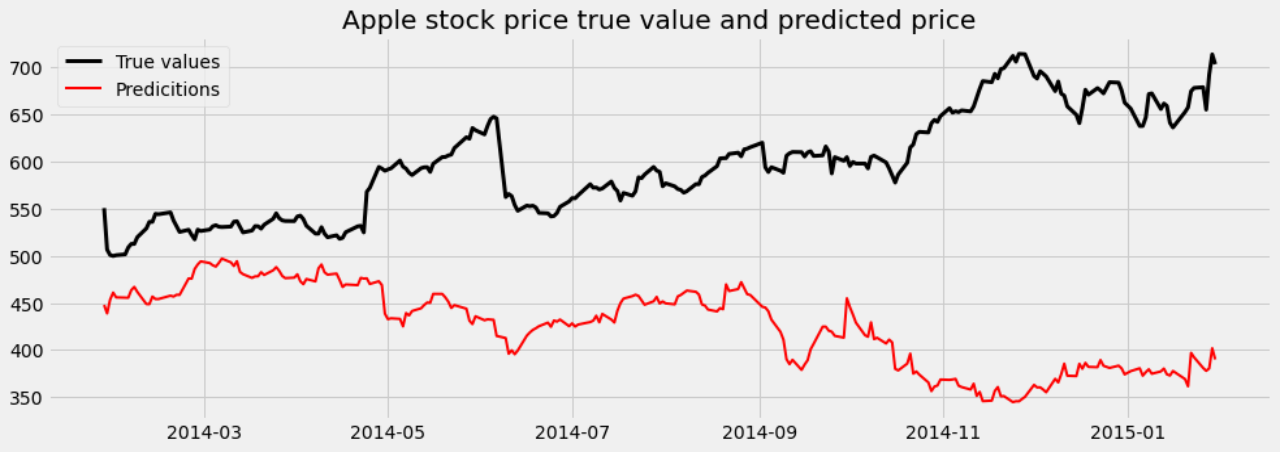


Figure 5.5. The predicted Apple stock prices (red) and the true Apple value (black).

Here, we visualize the predictions from several different models fit on the same data. We'll use Ridge regression, which has a parameter called "alpha" that causes coefficients to be smoother and smaller, and is useful if you have noisy or correlated variables.

We loop through a few values of alpha, initializing a model with each one and fitting it to the training data. We then plot the model's predictions on the test data, which lets us see what each model is getting right and wrong.

```
alphas = [.1, 1e2, 1e3]
ax.plot(y_test, color='k', alpha=.3, lw=3)

for ii, alpha in enumerate(alphas):
      y_predicted = Ridge(alpha=alpha).fit(X_train,
      y_train).predict(X_test)
      ax.plot(y_predicted, c=cmap(ii / len(alphas)))
      ax.legend(['True values', 'Model 1', 'Model 2', 'Model 3'])
      ax.set(xlabel="Time")
```

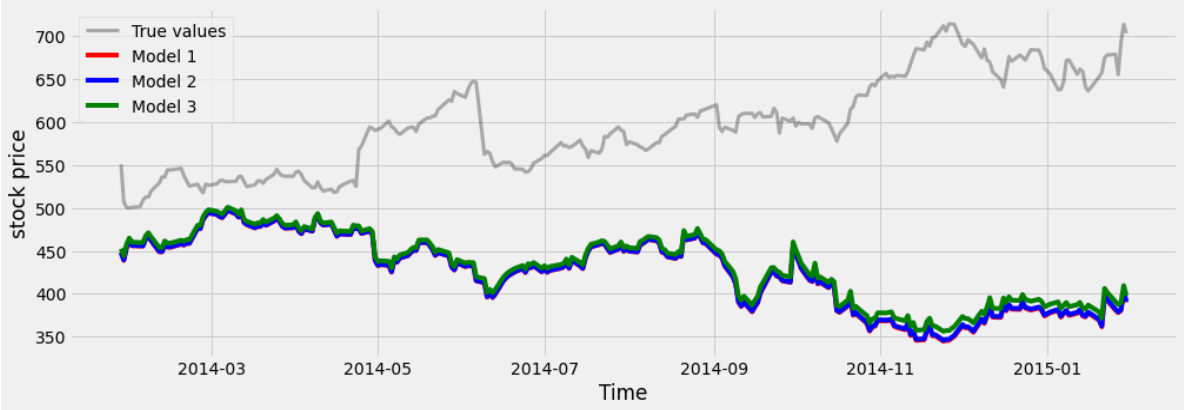


Figure 5.6. The True Apple stock prices (grey) and the predicted Apple stock prices with different alpha (red, blue, green).

2.2. Advanced time series forecasting

Now that we've covered some simple visualizations and model fitting with continuous time series, let's see what happens when we look at more real-world data. Real-world data is always messy and requires preparing and cleaning the data before fitting models. In time series, messy data often happens due to failing sensors or human error in logging the data.

Let's cover some specific ways to spot and fix messy data with time series. The two most common data problems with time-series data are **missing data and outliers.** First, let's create some messy-looking data. Since the original data of the stock prices has no missing data, we will randomly remove some parts of the data.

The data will be used is the stock market value of the American International Group (AIG) over the last several years. A consecutive percentage of the data will be removed using the function below:

```
# Create missing rows at random
def remove_n_consecutive_rows(frame, n, percent):
chunks_to_remove = int(percent/100*frame.shape[0]/n)

# split the indices into chunks of length n+2
      chunks = [list(range(i,i+n+2)) for i in range(0, frame.shape[0]-n)]
      drop_indices = list()

for i in range(chunks_to_remove):
        indices = random.choice(chunks)
        drop_indices+=indices[1:-1]
        # remove all chunks that contain overlapping values with indices
        chunks = [c for c in chunks if not any(n in indices for n in c)]

drop_indices = frame.index[drop_indices]
frame.iloc[drop_indices,] = np.nan
return frame
```

This function takes a data frame, the number of consecutive rows to be set to **NaN,** and the percentage of the data to be removed. In our case, the data frame is the AIG data, the number of consecutive rows to be removed is 100 rows, and the percentage of the data is 20%.
Since the data has 1278 rows, removing 20% of it means removing 255; therefore, two consecutive rows of 100 will be set to NaN. Let's plot the AIG data without any missing values, and after that, the AIG data with the missing values in Figures 5.7 and 5.8.

```
# The AIG data without missing values
AIG = pd.DataFrame(preprocessed_prices['AIG'])
AIG.plot(figsize=(8,8))

plt.title('The stock prices of the AIG company')
plt.xlabel('Time (Years)')
plt.ylabel('Stock price')
```

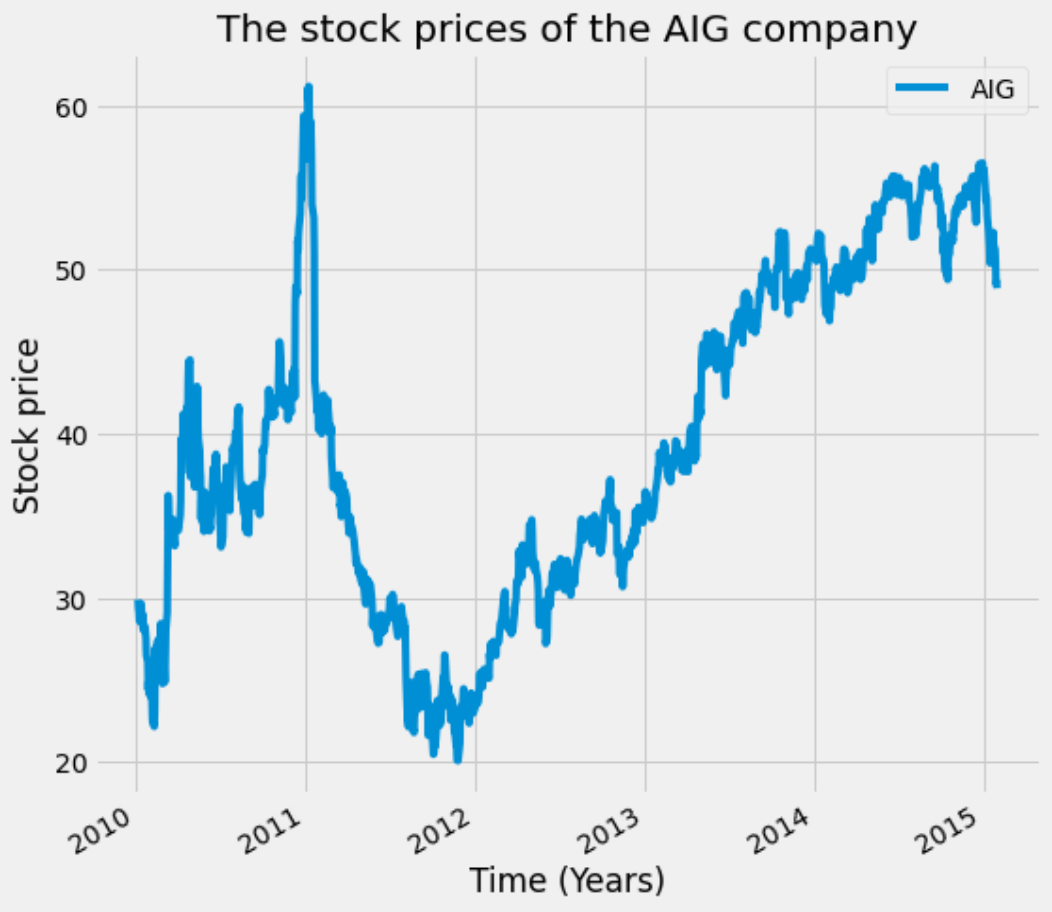


Figure 5.7. The AIG stock prices without missing data.

```
# plotting the missing data
AIG_missing_data = remove_n_consecutive_rows(AIG, 100, 20)

AIG_missing_data.plot(figsize=(8,8))
plt.title('The stock prices of the AIG company with missing data')
plt.xlabel('Time (Years)')
plt.ylabel('Stock price')
```

Let's now see how we can solve the problem with the missing data in our time series. We can first use a technique called **interpolation**, which uses the values on either end of a missing window of time to infer what's in between.

We will first create a Boolean mask that we'll use to mark where the missing values are. Next, we call the dot-interpolate method to fill in the missing values. We'll use the first argument to signal we want linear interpolation. Finally, we'll plot the interpolated values in Figure 5.9.

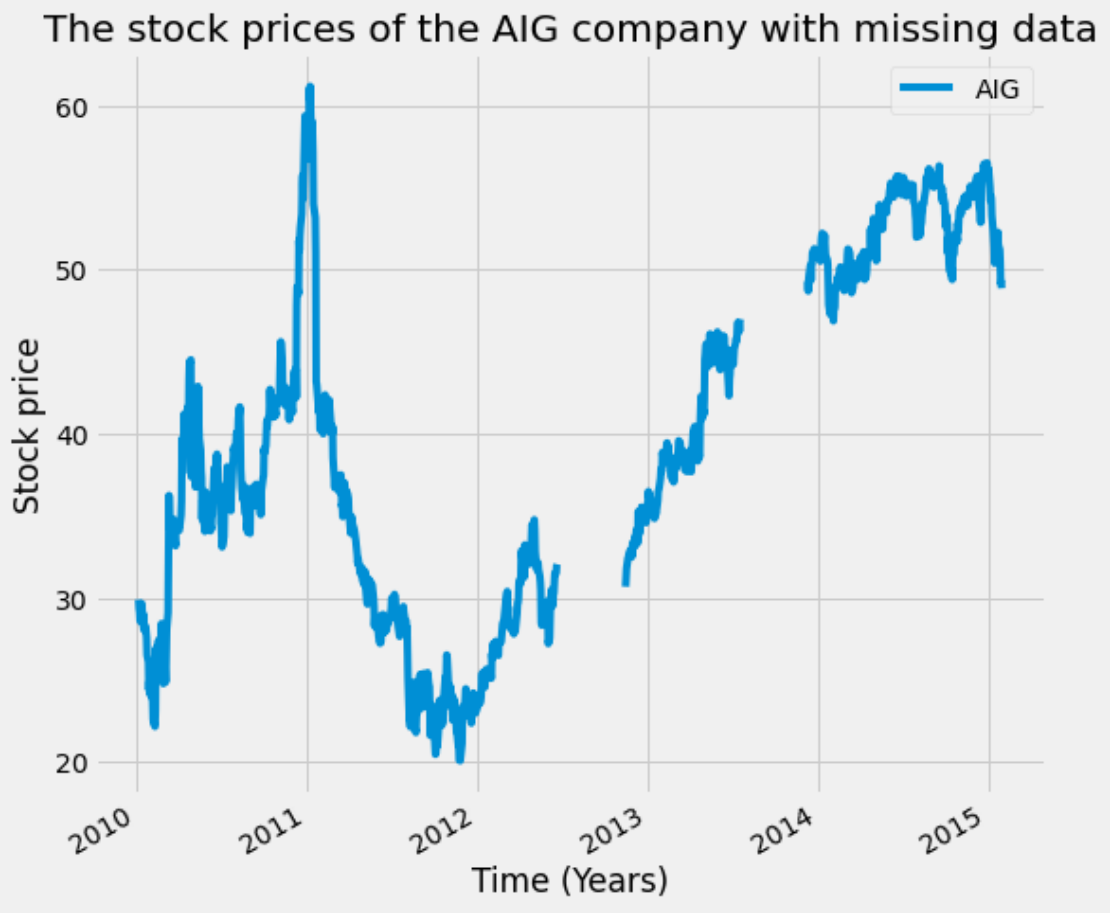


Figure 5.8. The AIG stock prices with randomly missing data.

```
# Interpolation in Pandas

# Return a boolean that notes where missing values are
missing_index = AIG_missing_data.isna()

# Interpolate linearly within missing windows
AIG_interp = AIG_missing_data.interpolate('linear')

# Plot the interpolated data in red and the data w/ missing values in
black
ax = AIG_interp.plot(c='r')
AIG_missing_data.plot(c='k',ax=ax, lw=2)
ax.legend(['AIG_missing','AIG_interpolated'])
ax.set(xlabel='Time (Years)')
ax.set(ylabel='Stock value')
```

```
ax.set(title='AIG stock price value with missing and interpolated
values')
```

Figure 5.9. AIG time series with missing data (red) and interpolated version(black).

You can see the results of interpolation in red. In this case, we used the "linear" argument, so the interpolated values are a line between the start and stop points of the missing window. Other arguments to the dot-interpolate method will result in different behavior.

Another common technique to preprocess the time series data is to transform it so that it is a more well-behaved version. To do this, we'll use the rolling window technique.
Using a rolling window, we'll calculate each time point's percent change over the mean of a window of previous time points. This standardizes the variance of our data and reduces long-term drift.

In the function below, we first separate the final value of the input array. Then, we calculate the mean of all but the last data point. Finally, we subtract the mean from the final data point and divide it by the mean. The result is the percent change for the final value.

```
# Transforming to percent change with Pandas

def percent_change(values):
    """ Calculates the % change between the last value
    and the mean of previous values"""
    # Separate the last value and all previous values into variables
    previous_values = values[:-1]
    last_value = values[-1]
    # Calculate the % difference between the last value
    # and the mean of earlier values
    percent_change = (last_value - np.mean(previous_values)) \
    / np.mean(previous_values)
    return percent_change
```

After that, we can apply this to our data using the dot-aggregate method, passing our function as an input. We plot the data, and as you can see on the right, the data is now roughly centered at zero, and periods of high and low changes are easier to spot.

```
# Applying the transformation to our data

# Plot the raw data
fig, axs = plt.subplots(1, 2, figsize=(20, 10))

# plot the AIG with interplotation
axs[0].plot(AIG_interp, label='AIG')
axs[0].legend()

# Calculate % change and plot
AIG_perc_change =
AIG_interp.rolling(window=20).aggregate(percent_change)

# plot the trasnfoemd AIG
axs[1].plot(AIG_perc_change,label= 'AIG_transfomred')
axs[1].legend()

# set the title and x-axis and y-axis labels
axs[0].set(xlabel='Time (Years)')
axs[1].set(xlabel='Time (Years)')
axs[0].set(ylabel='Stock market value')
axs[1].set(ylabel='Percentage chnage in the stock market value')
plt.suptitle('AIG stock prices vs percentage change in stock prices')
```

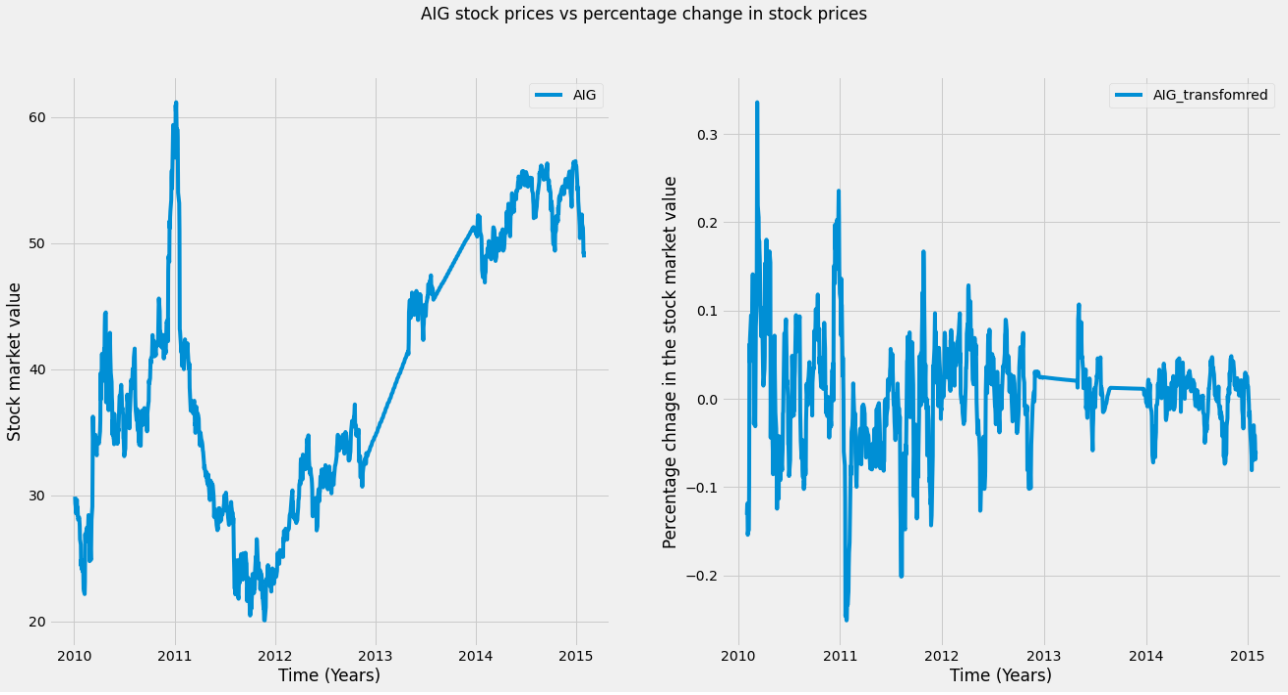


Figure 5.10. AIG stock prices with interpolation (left) and the percentage change in the AIG stock prices (right).

This transformation will be used to detect outliers. Outliers **are data points that are statistically different from the dataset as a whole.** A common definition is any data point that is more than three standard deviations away from the mean of the dataset.

We will visualize this definition of an outlier in the coming plot. We will calculate the mean and standard deviation of each dataset, then plot outlier "thresholds" (three times the standard deviation from the mean) on the raw and transformed AIG time series.

```
# Plotting a threshold on our data
fig, axs = plt.subplots(1, 2, figsize=(20, 10))
legends = ['AIG','AIG_transfromed']
for data, ax, l in zip([AIG_interp, AIG_perc_change], axs, legends):
    # Calculate the mean/standard deviation for the data
    data_mean = data.mean()
    data_std = data.std()
    # Plot the data, with a window that is 3 standard deviations
    # around the mean
    ax.plot(data, label=l)
    ax.legend()
    ax.axhline(data_mean[0] + data_std[0] * 3, ls='--', c='r')
    ax.axhline(data_mean[0] - data_std[0] * 3, ls='--', c='r')

# set the title and x-axis and y-axis labels
axs[0].set(xlabel='Time (Years)')
axs[1].set(xlabel='Time (Years)')
axs[0].set(ylabel='Stock market value')
axs[1].set(ylabel='Percentage chnage in the stock market value')
plt.suptitle('AIG stock prices vs percentage change in stock prices
with applied threshold')
```

As we can see in Figure 5.11, the left plot, there are no outliers, while in the right plot (transformed data), there are outliers (any datapoint outside these bounds could be an outlier). Note that the data points deemed an outlier depend on the transformation of the data.We will replace the outliers with the median of the remaining values. We first center the data by subtracting its mean and calculating the standard deviation.

Finally, we calculate the absolute value of each data point and mark any data point that lies outside of three standard deviations from the mean. We then replace these using the **nanmedian** function, which calculates the median without being hindered by missing values.

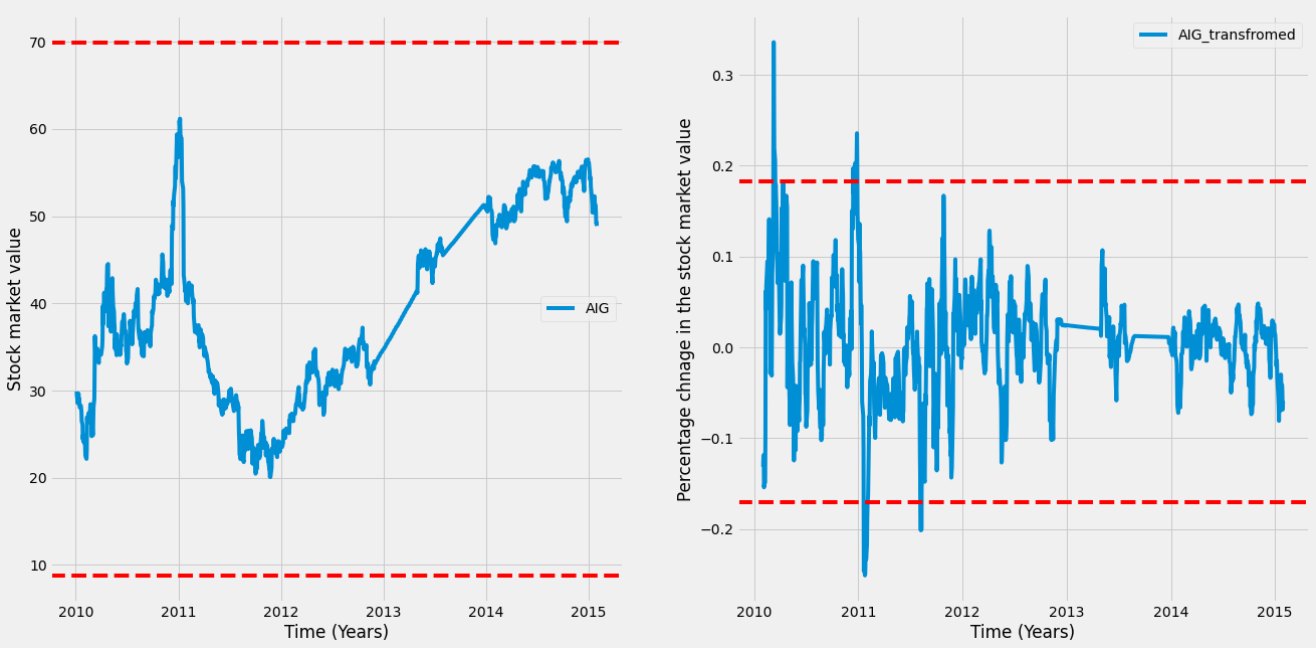


Figure 5.11. AIG stock prices vs percentage change in stock prices with applied threshold.

```
# Center the data so the mean is 0
AIG_outlier_centered = AIG_perc_change - AIG_perc_change.mean()

# Calculate standard deviation
std = AIG_perc_change.std()

# Use the absolute value of each datapoint
# to make it easier to find outliers
outliers = np.abs(AIG_outlier_centered) > (std * 3)

# Replace outliers with the median value
# We'll use np.nanmean since there may be nans around the outliers
AIG_outlier_fixed = AIG_outlier_centered.copy()
AIG_outlier_fixed[outliers] = np.nanmedian(AIG_outlier_fixed)
```

Finally, we will plot the transformed data after removing the outliers. As you can see, once we've replaced the outliers, there don't seem to be as many extreme data points. This should help our model find the patterns we want.

```
fig, axs = plt.subplots(1, 2,sharey=True,figsize=(20, 10))
axs[0].plot(AIG_outlier_centered, label='AIG with outliers')
axs[1].plot(AIG_outlier_fixed, label='AIG without outliers')

axs[0].legend()
axs[1].legend()

axs[0].set(xlabel='Time (Years)')
axs[1].set(xlabel='Time (Years)')
```

```
plt.suptitle('AIG stock prices with outliers Vs AIG stock prices
without outliers ')
axs[0].set(ylabel='Percentage chnage in the stock market value')
```

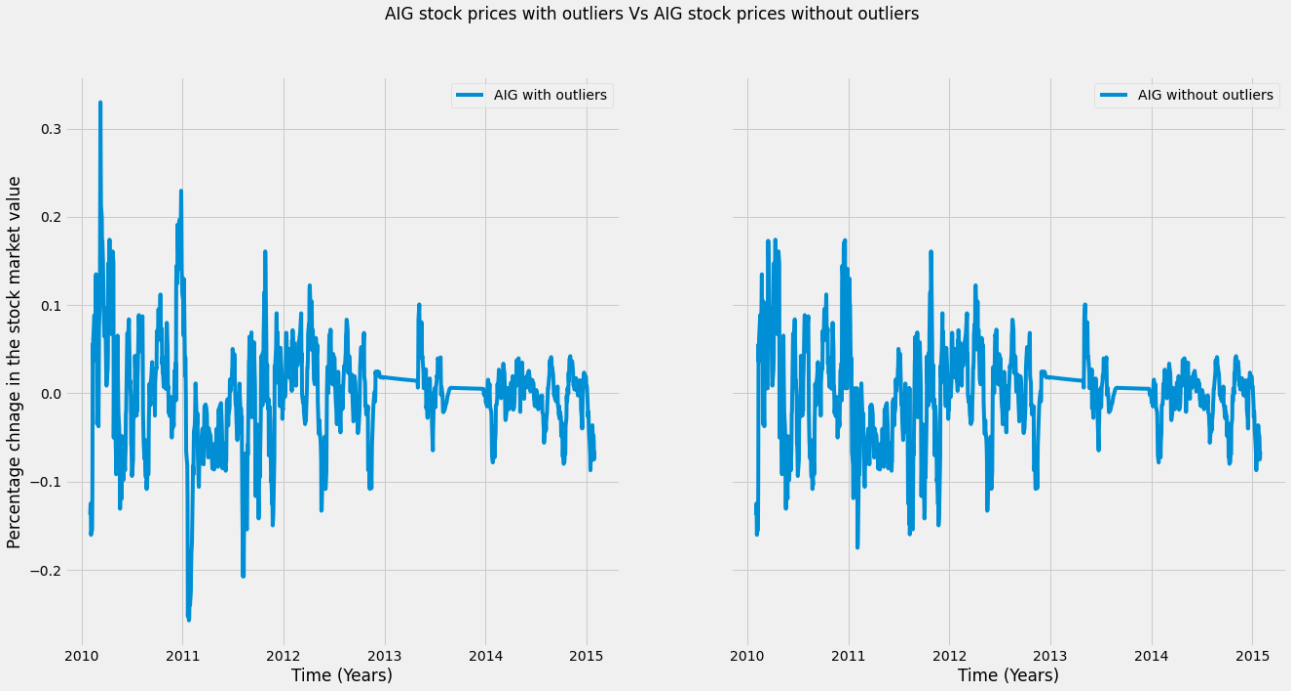


Figure 5.12. The AIG with outliers (left) and after removing them (right).

2.3. Creating features over time

In the final subsection of the second section of this article, we will extract specific features that are useful in time series analysis. First, we can calculate statistical properties such as standard deviation (std), mean, max, min, and so on for each window of our time series and use them as representative features for our time series.

We can define multiple functions of each window to extract many features at once using pandas. The **.aggregate()** method can be used to calculate many features of a window at once by passing a list of functions to the method, and each function will be called on the window and collected in the output. Below is an example of this.

We first use the **.rolling()** method to define a rolling window, then pass a list of two functions (for the standard deviation and maximum value). This extracts two features for each column over time.

```
# Calculate a rolling window, then extract two features
feats = preprocessed_prices.rolling(20).aggregate([np.std,
np.max]).dropna()
feats.head(5)
```

	AAPL		ABT		AIG		AMAT		ARNC		...	VZ		WFC	
	std	amax	std	amax	std	amax	std	amax	std	amax	...	std	amax	std	amax
date															
2010-02-01	6.536228	215.039995	0.868830	56.239949	2.051966	29.889999	0.815610	14.87	1.557116	17.450013	...	1.149069	33.339868	0.595400	29.129999
2010-02-02	6.776924	215.039995	0.869197	56.239949	2.101032	29.629999	0.834197	14.87	1.541017	17.450013	...	1.061109	33.339868	0.571024	29.129999
2010-02-03	6.655202	215.039995	0.852509	56.239949	2.157249	29.629999	0.862663	14.87	1.545563	17.450013	...	0.958873	31.919873	0.569911	29.129999
2010-02-04	7.148351	215.039995	0.873895	56.239949	2.282004	29.629999	0.916897	14.87	1.511028	17.450013	...	1.016933	31.909874	0.631350	29.129999
2010-02-05	7.265069	215.039995	0.891497	56.239949	2.400626	29.629999	0.931594	14.87	1.463785	17.450013	...	1.049564	31.909874	0.618206	28.990000

5 rows × 92 columns

You can extract many different kinds of features this way. It is always good to plot the features you've extracted over time, as this can give you a clue about how they behave and help you spot noisy data and outliers. **In Figure 5.13, we can see that the maximum value is much higher than the mean**.

```
feats['AAPL'].plot(figsize=(10, 10))
plt.xlabel('Date [Years] ')
plt.ylabel('Stock price')
plt.title('The std Vs max of the stock price change of Apple')
```

A useful tool when using the dot-aggregate method is the partial function. This is built into Python and lets you create a **new** function from an old one, with some of the parameters pre-configured.

In this example, we first import **partial** from the **functools** package, then use it to create a mean function that always operates on the first axis. The first argument is the function we want to modify, and subsequent key-value pairs will be preset in the output function. **After this, we no longer need to configure those values when we call the new function.**

```
# If we just take the mean, it returns a single value
a = np.array([[0, 1, 2], [0, 1, 2], [0, 1, 2]])
print(np.mean(a))

# We can use the partial function to initialize np.mean
# with an axis parameter
from functools import partial
mean_over_first_axis = partial(np.mean, axis=0)
print(mean_over_first_axis(a))
```

The output is: [0 \| 1\| 2], which is the mean of each column.

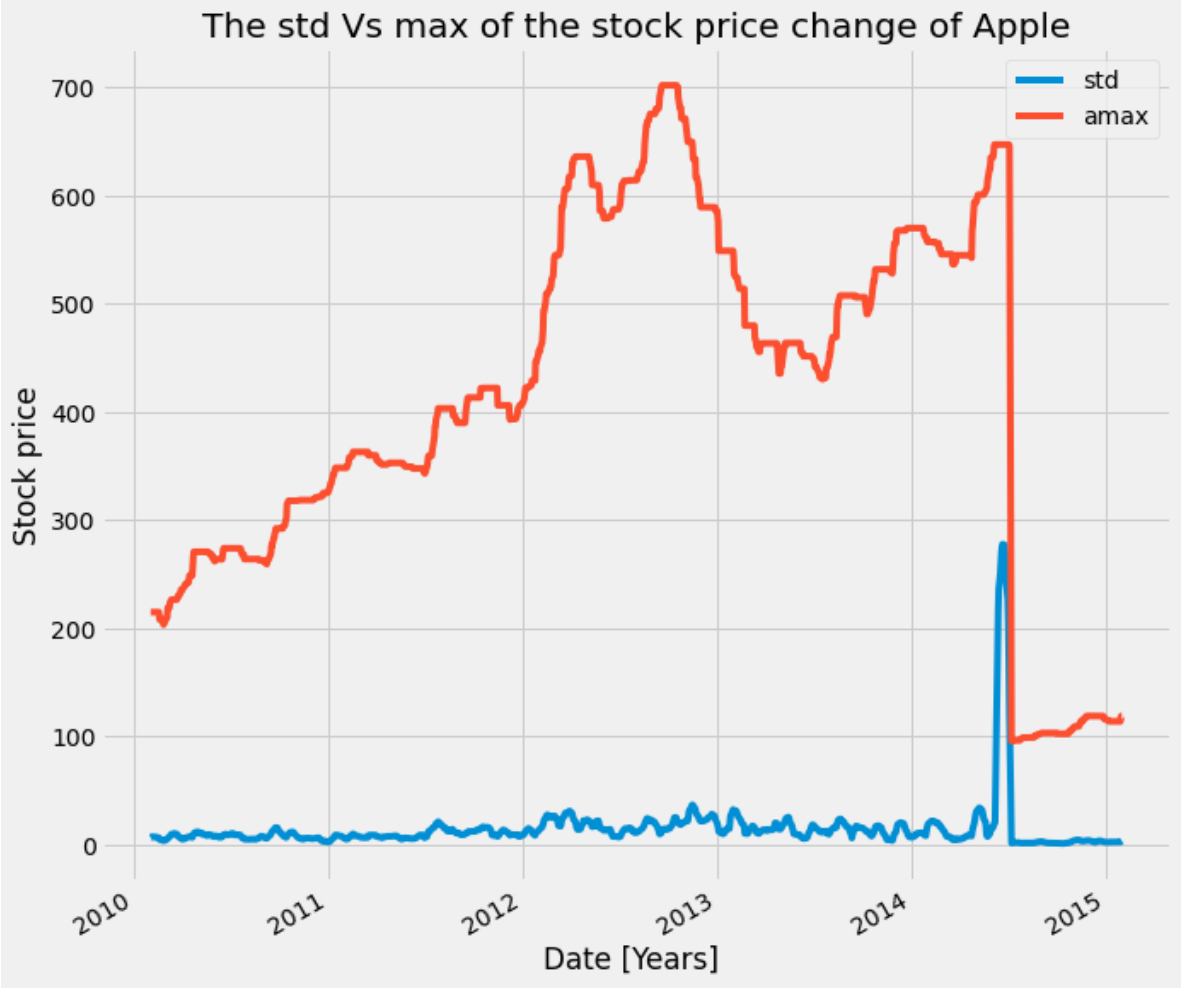


Figure 5.13. The std vs the maximum of the Apple stock prices.

Let's go back to feature extraction from time-series data. A particularly useful tool for feature extraction is the percentile function. This is similar to calculating the mean or median of your data, but it gives you more fine-grained control over what is extracted.

For a given dataset, the Nth percentile is the value where N% of the data is below that data point, and 100-N% of the data is above that data point. In the example below, the **percentile** function takes an array as the first input and an integer between 0 and 100 as the second input.

It will return the value in the input array that matches the percentile you've chosen. Here it returns 40, which means that the value "40" is larger than 20% of the input array.

```
# Percentiles summarize your data
print(np.percentile(np.linspace(0, 200), q=20))
```

The output: 400

Here we'll combine the **percentile** function with **partial** functions to extract several percentiles with the **.aggregate()** method. We use a list comprehension to create a list of functions (called

percentile_funcs). Then, we loop through the list, calling each function on our data, to return a different percentile of the data.

We will first apply it to test data to understand how it works. The test data is created using **np.linspace** to create evenly spaced numeric sequences and then we will create limit to it.

```
# Combining np.percentile() with partial functions to calculate a
range of percentiles and apply it on toy data

data = np.linspace(0, 100)

# Create a list of functions using a list comprehension
percentile_funcs = [partial(np.percentile, q=ii) for ii in [20, 40,
60]]

# Calculate the output of each function in the same way
percentiles = [i_func(data) for i_func in percentile_funcs]
print(percentiles)
```

Output: [20.0, 40.00000000000001, 60.0]

Then we could pass this list of partial functions to our **.aggregate()** method to extract several percentiles for each column of our time series data.

```
# Calculate multiple percentiles of a rolling window on our prices
time series

preprocessed_prices.rolling(20).aggregate(percentile_funcs).dropna()
```

	AAPL			ABT			AIG			AMAT	...	WFC	XOM	
	percentile	percentile	percentile	percentile	percentile	percentile	percentile	percentile	percentile	percentile	...	percentile	percentile	percentile
date														
2010-02-01	202.314001	207.816003	210.298002	54.005953	54.497952	55.097952	24.808000	27.795999	28.507999	12.638000	...	28.340001	65.905998	67.497998
2010-02-02	198.981996	207.008001	209.702001	54.005953	54.497952	55.097952	24.366000	27.129999	28.334000	12.626000	...	28.438000	65.905998	66.855998
2010-02-03	198.933995	205.936001	208.613998	54.245953	54.593951	55.097952	24.216000	26.336000	28.135999	12.592000	...	28.438000	65.905998	66.659997
2010-02-04	197.371999	204.786001	207.956001	53.939952	54.593951	55.097952	24.122000	25.618000	27.999999	12.518000	...	28.438000	65.787999	66.431999
2010-02-05	195.779998	201.557999	207.784003	53.939952	54.497952	55.097952	23.967999	24.706000	27.713999	12.464000	...	28.340001	65.424001	66.147999
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
2015-01-26	107.564000	109.462003	110.983998	44.206001	44.490002	44.904001	51.189999	52.115999	54.228001	23.441999	...	53.379999	90.232001	91.009997
2015-01-27	107.564000	109.298001	110.283999	44.132002	44.454002	44.816000	51.054001	51.725999	53.680000	23.441999	...	53.224000	90.232001	90.925998
2015-01-28	107.564000	109.298001	110.283999	44.038001	44.352001	44.670000	50.794002	51.457999	52.744001	23.369999	...	53.062001	89.962000	90.822000
2015-01-29	107.564000	109.298001	110.283999	44.038001	44.352001	44.670000	50.614002	51.312000	52.312000	23.369999	...	52.843999	89.795998	90.564001
2015-01-30	107.564000	109.298001	110.888000	44.038001	44.352001	44.670000	50.456001	51.170000	52.033999	23.330000	...	52.711999	89.583998	90.314002

1259 rows × 138 columns

We can see that for each column, the three percentiles functions were applied to our data with a rolling window of 20. Another common feature to consider is the "**date-based**" features. That is, features that take into consideration information like "what time of the year is it?" or "is it a holiday?".

Since many datasets involve humans, these pieces of information are often important. For example, if you're trying to predict the number of customers that will visit your store each day, it's important to know if it's the weekend or not!

Working with dates and times is straightforward in Pandas. Datetime functionality is most commonly accessed with a DataFrame's index. You can use the **to_datetime** function to ensure dates are treated as DateTime objects. You can also extract many date-specific pieces of information, such as the day of the week or weekday name, as shown below. These could then be treated as features in your model.

```
# datetime features using Pandas
# Ensure our index is datetime
preprocessed_prices.index = pd.to_datetime(preprocessed_prices.index)

# Extract datetime features
day_of_week_num = preprocessed_prices.index.weekday
print('Days of the week in numbers:', day_of_week_num[:10])

day_of_week = preprocessed_prices.index.day_name()
print('Days of the week in names:', day_of_week[:10])
```

```
Days of the week in numbers: Int64Index([0, 1, 2, 3, 4, 0, 1, 2, 3, 4], dtype='int64', name='date')
Days of the week in names: Index(['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Monday',
       'Tuesday', 'Wednesday', 'Thursday', 'Friday'],
      dtype='object', name='date')
```

3. Evaluation and Inspecting Time Series Models

Once you've got a model for predicting time series data, you need to decide if it's a good or a bad model. This section covers the basics of generating predictions with models in order to validate them against "test" data.

3.1. Creating features from the past

Perhaps the biggest difference between time series data and "non-time series" data is the relationship between data points. Because the data has a linear flow (matching the progression of time), patterns will persist over a span of data points. As a result, we can use information from the past in order to predict values in the future.

It's important to consider how **smooth** your data is when fitting models with time-series data. The smoothness of your data reflects how much correlation there is between a one-time point and those that come before and after it.

The extent to which previous time points are predictive of subsequent time points is often described as **autocorrelation** and can have a big impact on the performance of your model.

Let's investigate this by creating a model in which previous time points are used as input features to the model. Remember that the regression models will assign a **weight** to each input feature, and we can use these weights to determine how **smooth** or **autocorrelated** the time series is.

First, we'll create time-shifted versions of our data. This entails "rolling" your data either into the future or into the past so that the same index of data now has a different time point in it. We can do this in Pandas by using the dot-shift method of a DataFrame. Positive values roll the data backward, while negative values roll the data forward.

Here, we use a dictionary comprehension that creates several time-lagged versions of the data. Each one shifts the data to a different number of indices from the past.
Since our data is recorded daily, this corresponds to shifting the data so that each index corresponds to the value of the data N days prior. We can then convert this into a DataFrame where dictionary keys become column names and fill in the missing values in each lag with zero.

```
# create shifts in the data

# slice the AIG company data
data = preprocessed_prices['AIG']
shifts = [1, 2, 3, 4, 5, 6, 7]

# Create a dictionary of time-shifted data
many_shifts = {'lag_{}'.format(ii): data.shift(ii) for ii in shifts}

# Convert the shifts into a dataframe
many_shifts = pd.DataFrame(many_shifts)
many_shifts.fillna(0, inplace=True)
many_shifts.head(15)
```

We will now fit a scikit-learn regression model. Note that in this case, "many_shifts" is simply a time-shifted version of the time series contained in the **data** variable. We'll fit the model using Ridge regression, which spreads out weights across features.

```
# Fit the model using these input features
model = Ridge()
model.fit(many_shifts, data)
```

Once we fit the model, we can investigate the coefficients it has found. **Larger absolute values of coefficients mean that a given feature has a large impact on the output variable**. We can use a bar plot in Matplotlib to visualize the model's coefficients that were created after fitting the model.

```
# Visualize the fit model coefficients
fig, ax = plt.subplots(figsize=(10,10))
ax.bar(many_shifts.columns, model.coef_)
ax.set(xlabel='Coefficient name', ylabel='Coefficient value')

# Set formatting so it looks nice
```

```
plt.setp(ax.get_xticklabels(), rotation=45,
horizontalalignment='right')
plt.title('Model coefficients for each lag')
```

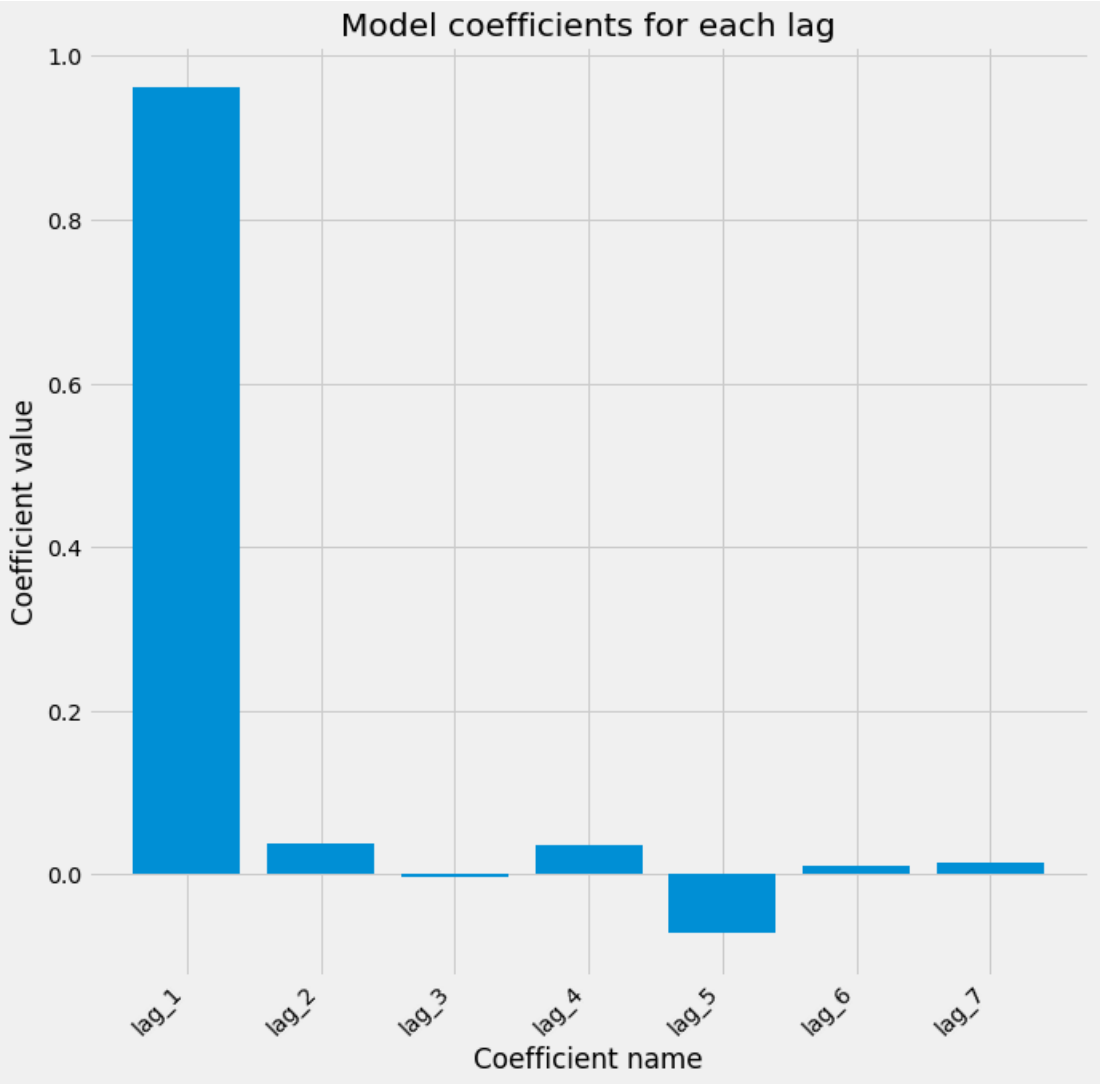


Figure 5.14. Model coefficients for each lag.

3.2. Cross-validating time-series data

The most common form of cross-validation is k-fold cross-validation. In this case, data is split into K subsets of equal size. In each iteration, a single subset is left out as the validation set and the model is trained using the rest k-1 folds. Here we show how to initialize a k-fold iterator with scikit-learn.

```
from sklearn.model_selection import KFold
cv = KFold(n_splits=5)
for tr, tt in cv.split(X, y):
  ...
```

Many cross-validation iterators let you randomly shuffle the data. This may be appropriate if your data is independent and identically distributed, but time-series data is usually not i.i.d. Here we use the "**ShuffleSplit**" cross-validation iterator, which randomly permutes the data labels in each iteration. Let's see what our visualization looks like when using shuffled data.

The data used is a subset of the **stock data**, where the training data is the Yahoo, Nvidia, and eBay stock data, while the target value is the Apple stock data.

```
from sklearn.model_selection import ShuffleSplit

cv = ShuffleSplit(n_splits=10, random_state=1)

# convert the dataframe into numpy array
X = X.to_numpy()
y = y.to_numpy()


# Iterate through CV splits
results = []
for tr, tt in cv.split(X, y):
    # Fit the model on training data
    model.fit(X[tr], y[tr])

    # Generate predictions on the test data, score the predictions,
and collect
    prediction = model.predict(X[tt])
    score = r2_score(y[tt], prediction)
    results.append((prediction, score, tt))

# Custom function to quickly visualize predictions
visualize_predictions(results)
```

We can see from the Figures 5.15 and 5.16 that the validation indices are all over the place, not in chunks like before. That's because we used a cross-validation object that shuffles the data, so the predictions will not follow the time order of the signal.

We see that the output data no longer "looks" like a time series because the temporal structure of the data has been destroyed. If the data is shuffled, it means that some information about the training set now exists in the validation set, and you can no longer trust and rely on the score of your model.

If we set the shuffle argument to false, we can see that both will be the predictions ordered by time, and the predictions ordered by the prediction number are the same, because there is random shuffling occurring.

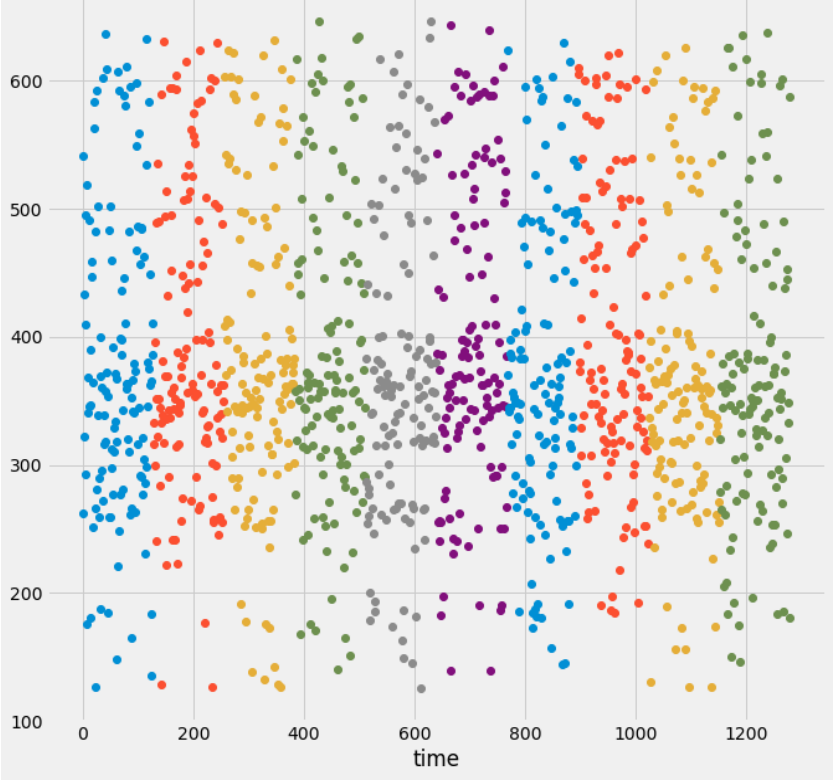


Figure 5.15. Predictions ordered by test prediction number.

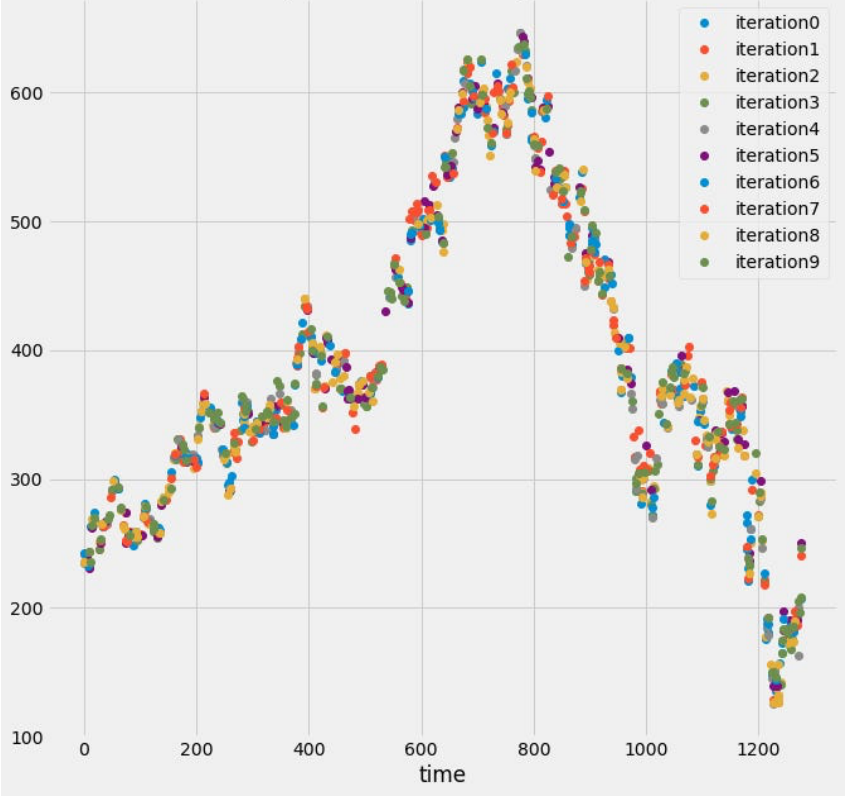


Figure 5.16. Predictions ordered by time.

```
# Create KFold cross-validation object
from sklearn.model_selection import KFold
cv = KFold(n_splits=10, shuffle=False)

# Iterate through CV splits
results = []
for tr, tt in cv.split(X, y):
    # Fit the model on training data
    model.fit(X[tr],y[tr])

    # Generate predictions on the test data and collect
    prediction = model.predict(X[tt])
    results.append((prediction, tt))

# Custom function to quickly visualize predictions
visualize_predictions(results)
```

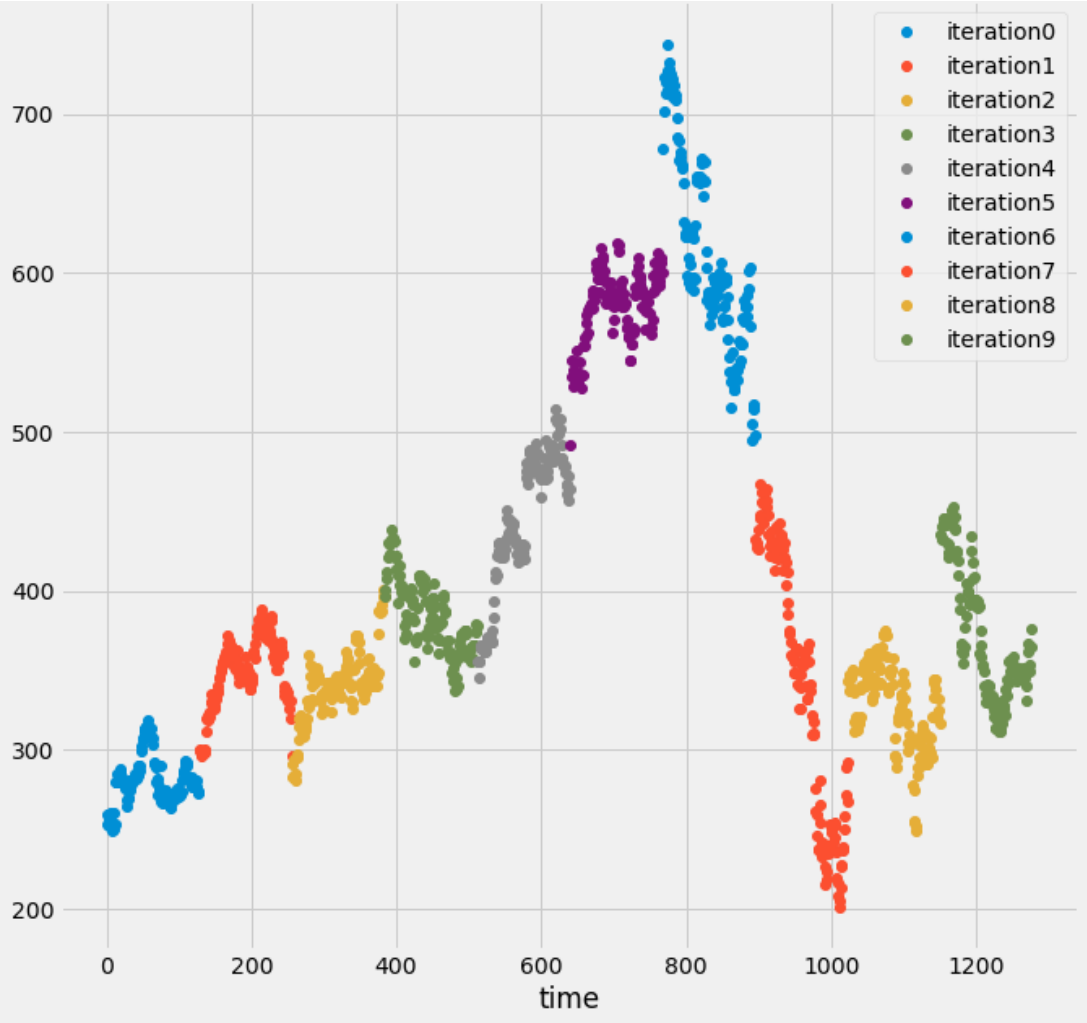


Figure 5.17. Predictions are ordered by time.

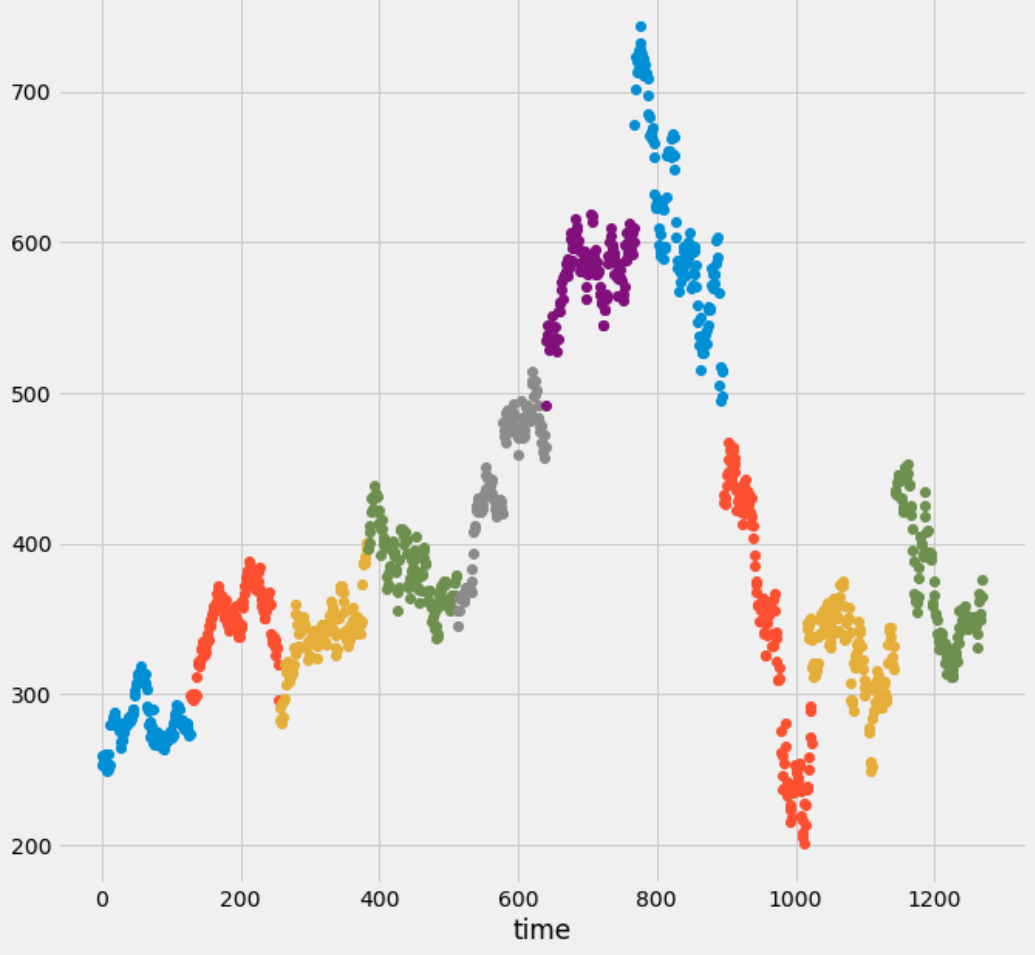


Figure 5.18. Predictions are ordered by test prediction number.

Finally, we'll cover another cross-validation iterator that can help deal with time-series data. There is one cross-validation technique that is meant particularly for time series data. This approach **always** uses data from the past to predict time points in the future. Through CV iterations, a larger amount of training data is used to predict the next block of validation data, corresponding to the fact that more time has passed.

This more closely mimics the data collection and prediction process in the real world. We can use this cross-validation approach with the TimeSeriesSplit object in scikit-learn.
We can also visualize the training data in each iteration of cross-validation. From the figure below, we can see that the size of the training set grew each time you used the time series cross-validation object. This way, the time points you predict are always *after* the time points we train on.

```
from sklearn.model_selection import TimeSeriesSplit

# Create time-series cross-validation object
cv = TimeSeriesSplit(n_splits=10)

# Iterate through CV splits
fig, ax = plt.subplots()
```

```
for ii, (tr, tt) in enumerate(cv.split(X, y)):
    # Plot the training data on each iteration, to see the behavior of
the CV
    ax.plot(tr, ii + y[tr]/1000)

ax.set(title='Training data on each CV iteration', ylabel='CV
iteration')
ax.set(xlabel='time')
plt.show()
```

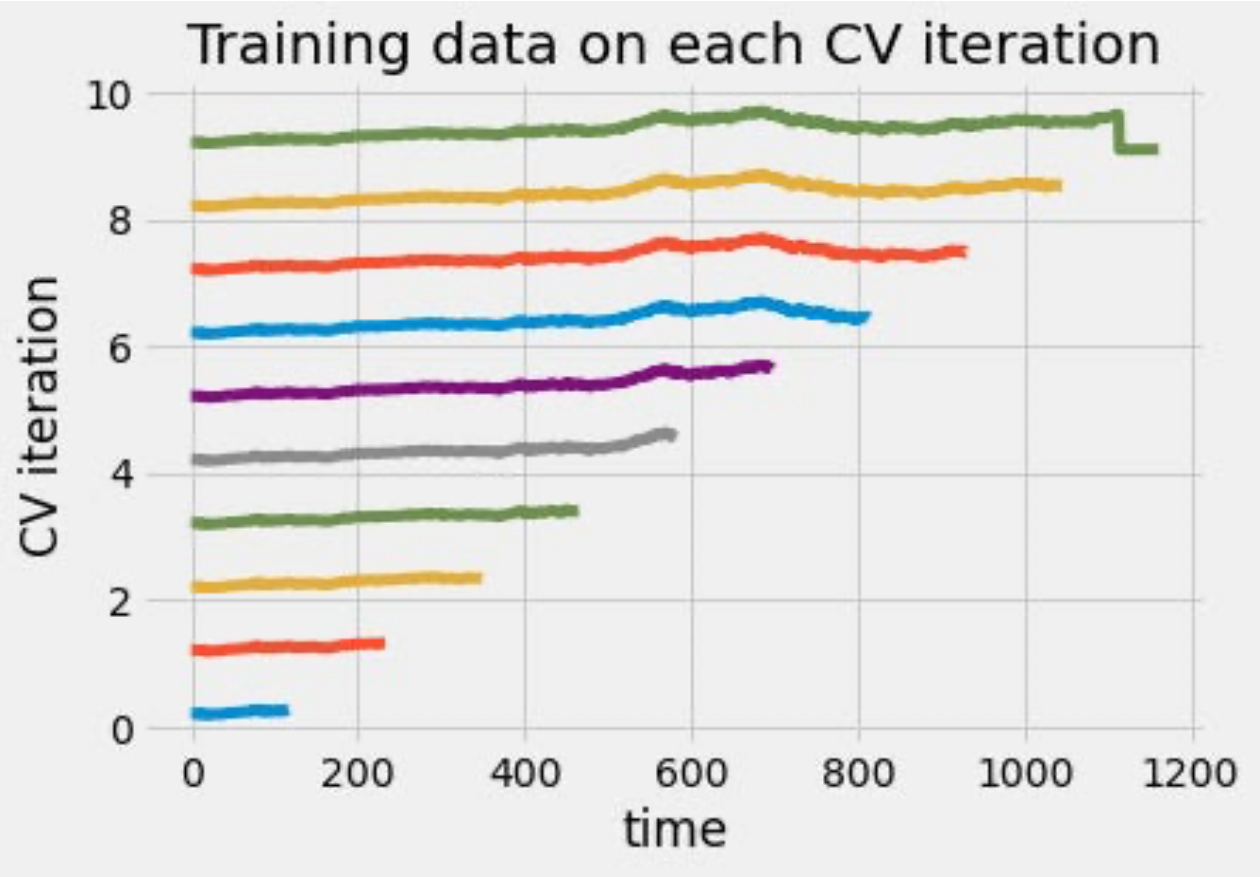


Figure 5.19. Training data on each CV iteration.

3.3. Stationarity and stability

A stationary signal is one that does not change its statistical properties over time. It has the same mean, standard deviation, and general trends. A non-stationary signal does change its properties over time.

Each of these has important implications for how to fit your model. Here's an example of a stationary and a non-stationary signal. On top, we see a highly non-stationary signal.
Its variance and trends change over time. At the bottom, we can see that the signal generally does not change its structure. Its variability is constant throughout time. Almost all real-world data are non-stationary.

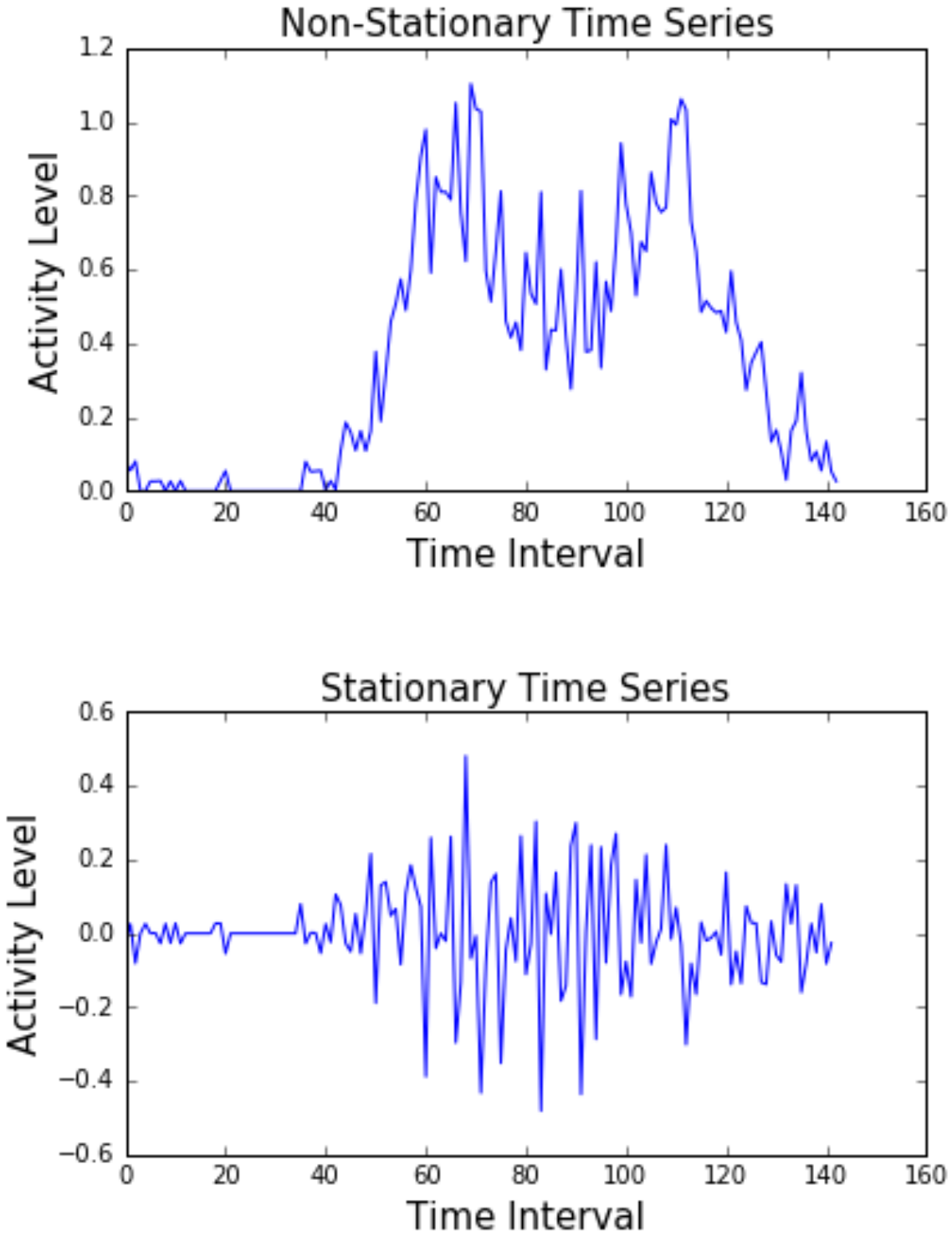


Figure 5.20. Stationary Vs non-stationary time series.

Most models have an implicit assumption that the relationship between inputs and outputs is static. If this relationship changes (because the data is not stationary), then the model will generate predictions using an outdated relationship between inputs and outputs. Therefore, it is important to know how to quantify and correct this when it occurs.

One approach is to use **cross-validation** as described in the previous subsection, which yields a set of model coefficients per iteration. We can quantify the variability of these coefficients across iterations. If a model's coefficients vary widely between cross-validation splits, there's a good chance the data is non-stationary (or noisy).

Another approach is using **bootstrapping. Bootstrapping** is a way to estimate the confidence in the mean of a collection of numbers. To perform a bootstrap for the mean, take many random samples (with replacement) from your collection of numbers and calculate the mean of each. Now, calculate lower/upper percentiles for this list. The lower and upper percentiles represent the variability of the mean.

Here's an example using scikit-learn and NumPy. Use the resample function in scikit-learn to take a random sample of coefficients, then use NumPy to calculate the mean for each coefficient in the sample and store it in an array.

Then, we calculate the 2-point-5 and 97-point-5 percentiles of the results to calculate the lower and upper bounds for each coefficient. This is called a 95% confidence interval.

```
# Bootstrapping the mean
from sklearn.utils import resample

# cv_coefficients has shape (n_cv_folds, n_coefficients)
n_coefficients = cv_coefficeints.shape[-1]
n_boots = 100
bootstrap_means = np.zeros((n_boots, n_coefficients))

for ii in range(n_boots):
    # Generate random indices for our data with replacement,
    # then take the sample mean
    random_sample = resample(cv_coefficeints)
    bootstrap_means[ii] = random_sample.mean(axis=0)

# Compute the percentiles of choice for the bootstrapped means
percentiles = np.percentile(bootstrap_means, (2.5, 97.5), axis=0)

# Plotting the bootstrapped coefficients
fig, ax = plt.subplots(figsize=(10,5))
ax.scatter(X_df.columns, percentiles[0], marker='_', s=200)
ax.scatter(X_df.columns, percentiles[1], marker='_', s=200)
ax.set(title='95% confidence intervals for model coefficients')
```

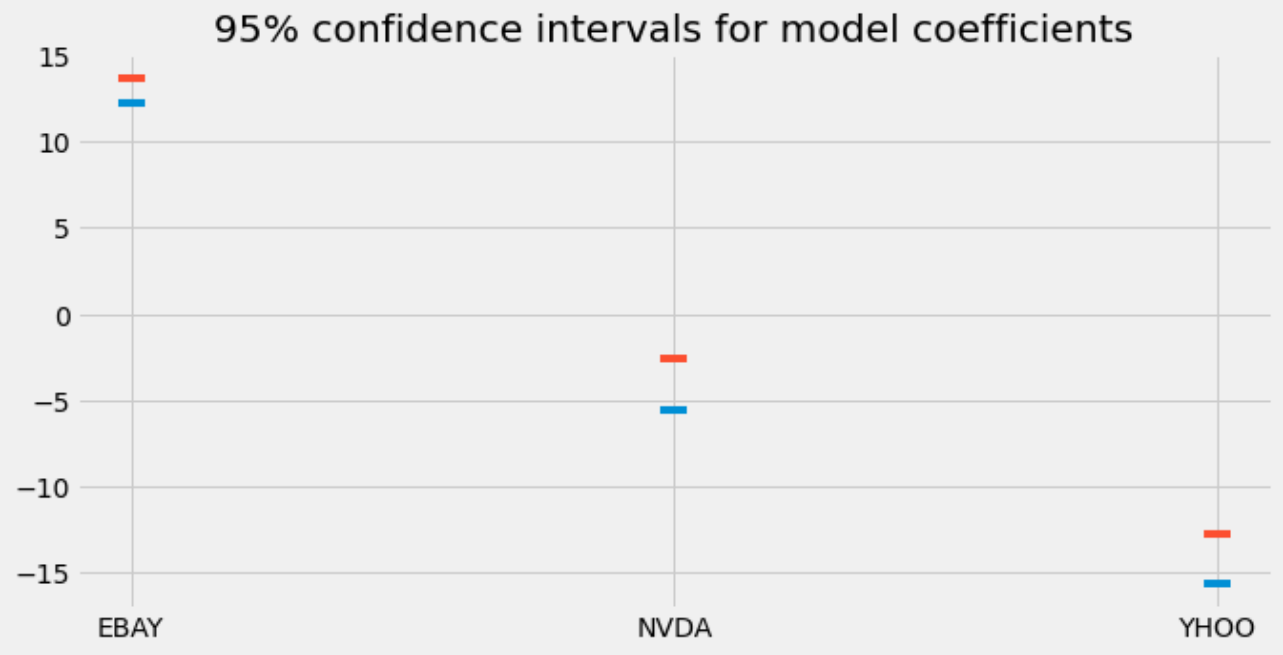


Figure 5.21. 95% confidence intervals for model coefficients.

Here we plot the lower and upper bounds of the 95% confidence intervals we calculated. This gives us an idea of the variability of the mean across all cross-validation iterations.
It's also common to quantify the stability of a **model's predictive power** across cross-validation folds. If you're using the TimeSeriesSplit object mentioned before, then you can visualize this as a time series.

This is useful in finding certain regions of time that hurt the score. Also useful for finding non-stationary signals. In the example below, we'll use the cross_val_score function, along with the TimeSeriesSplit iterator, to calculate the predictive power of the model over cross-validation splits.

We first create a small scoring function that can be passed to cross_val_score. Next, we use a list comprehension to find the date of the beginning of each validation block. Finally, we collect the scores and convert them into a Pandas Series.

Because the cross-validation splits happen linearly over time, we can visualize the results as a time series. If we see large changes in the predictive power of a model at one moment in time, it could be because the statistics of the data have changed. Here we create a rolling mean of our cross-validation scores and plot it with matplotlib.

```
# Model performance over time

# score function will be the correlation between the predicted and the
true values
def my_corrcoef(est, X, y):
    """Return the correlation coefficient
    between model predictions and a validation set."""
    score = np.corrcoef(np.hstack((y, est.predict(X))))[1, 0]
```

```
    return score

# define the cv split and the regression model
cv = TimeSeriesSplit(n_splits=100)
model = Ridge()
first_indices = []

# Grab the date of the first index of each validation set
for tr, tt in cv.split(X, y):
    # Fit the model on training data
    first_indices.append(X_df.index[tt[0]])

# Calculate the CV scores and convert to a Pandas Series
cv_scores = cross_val_score(model, X, y, cv=cv, scoring = my_corrcoef)
cv_scores = pd.DataFrame(cv_scores, index=first_indices)

# Visualizing model scores as a timeseries
fig, axs = plt.subplots(2, 1, figsize=(20, 20), sharex=False)

# Calculate a rolling mean of scores over time
cv_scores_mean = cv_scores.rolling(10, min_periods=1).mean()
cv_scores_mean.plot(ax=axs[0])
axs[0].set(title='Validation scores (correlation)', ylim=[0, 1])

# Plot the raw data
X_df.plot(ax=axs[1])
axs[1].set(title='Validation data')
```

In Figure 5.22, We can see the scores of our model across validation sets, which means over time, as we are splitting them linearly with time. There is a clear dip at the beginning of the data, probably because the statistics of the data changed.

One option is to restrict the size of the training window. This ensures that only the latest data points are used in training. We can control this with the max_train_size parameter.

```
# Only keep the last 100 datapoints in the training data
window = 100

# Initialize the CV with this window size
cv = TimeSeriesSplit(n_splits=10, max_train_size=window)

model = Ridge()
first_indices = []

for tr, tt in cv.split(X, y):
    # Fit the model on training data
    first_indices.append(X_df.index[tt[0]])


model = Ridge()
cv_scores = cross_val_score(model, X, y, cv=cv, scoring = my_corrcoef)
```

```
# Calculate the CV scores and convert to a Pandas Series
cv_scores = pd.DataFrame(cv_scores, index=first_indices)

#Visualizing model scores as a timeseries
fig, axs = plt.subplots(2, 1, figsize=(20, 20), sharex=False)

# Calculate a rolling mean of scores over time
cv_scores_mean = cv_scores.rolling(10, min_periods=1).mean()
cv_scores_mean.plot(ax=axs[0])
axs[0].set(title='Validation scores (correlation)', ylim=[0, 1])

# Plot the raw data
X_df.plot(ax=axs[1])
axs[1].set(title='Validation data')
```

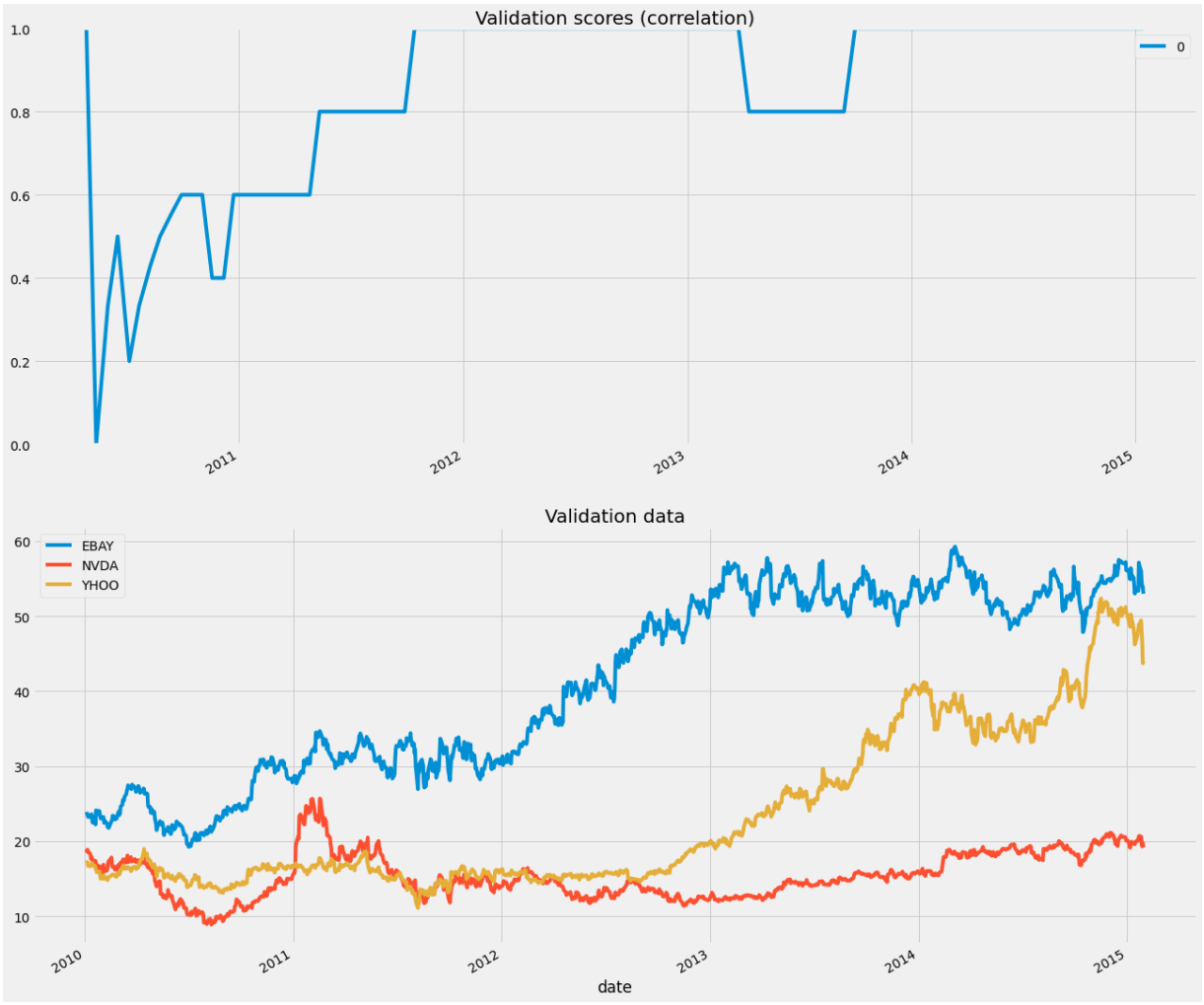


Figure 5.22. The validation set scores vs the validation data.

Conclusion

We have traveled a long way from the first page of this book. We began with simple timestamps and raw lists of numbers, and we have ended with sophisticated machine learning pipelines capable of predicting the future.

Throughout this journey, we have established that time series analysis is as much an art as it is a science. In Chapter 1, we learned that data preparation is not just a chore—it is the foundation of our success. By mastering resampling and window functions, we gained control over the temporal dimension. In Chapters 2 and 3, we learned to listen to the data, using statistical tests and visualizations to diagnose whether a signal was stationary, seasonal, or driven by random noise.

In Chapter 4, we explored the rigor of classical statistics. We saw that while ARIMA models require careful tuning and adherence to strict assumptions, they offer powerful, interpretable results for data with clear structures. Finally, in Chapter 5, we broke free of those strict assumptions. We embraced the flexibility of machine learning, learning that with the right feature engineering, we can model complex, non-linear relationships that traditional methods might miss.

The Road Ahead

You now possess a comprehensive toolkit. You know how to clean data, visualize trends, diagnose statistical properties, and choose between classical statistical models and modern machine learning approaches.

However, the field of time series analysis is vast and ever-evolving. As you move forward, consider exploring Deep Learning techniques, such as Long Short-Term Memory (LSTM) networks or Transformer models, which represent the next frontier in forecasting sequence data.

Remember the core lesson of this book: Trust your validation. Whether you are using a simple moving average or a complex regression model, the golden rule of time series is that we must never cheat by looking into the future during training. Stick to the rigorous cross-validation techniques you have learned here, and you will build models that survive the test of time. Thank you for reading, and happy forecasting.

Afterword

Thank you for trusting me with your time and choosing ***Hands-On Time Series Analysis with Python*** **as your guide.** I hope this book has demystified the complexities of temporal data and empowered you with the practical skills needed to build robust forecasting models.

Writing technical books is a continuous journey of learning, and I value your input immensely. Whether you have a specific question about a code snippet, feedback on a chapter, or simply want to share a success story about how you applied these techniques in your work, I would love to hear from you.

Get in Touch
You can reach me directly via email at: Youssef.Hosni95@outlook.com

Let's Stay Connected
Data science is better when we learn together. I invite you to join my professional network and follow my latest work through the channels below:

- **LinkedIn:** Connect with me personally to discuss industry trends and career growth.
- **Medium:** Follow me for in-depth articles and tutorials that go beyond the scope of this book.
- **Newsletter:** Subscribe to receive exclusive insights, updates on my future projects, and new writing directly to your inbox.
- **My Website:** Visit to explore my other books and resources.

If you found this book helpful, please consider leaving a review or sharing it with a colleague. Your support helps me continue creating content for the community.

Happy Forecasting!

Youssef Hosni

What's inside the book?

- Manipulating Time Series Data
- Time Series Data Analysis In Python
- Visualizing Time Series Data
- Time Series Forecasting with ARIMA Models Time Series Forecasting with Machine Learning

About the Author

Youssef Hosni is a data scientist and machine learning researcher who has worked in machine learning and AI for over half a decade.
In addition to being a researcher and data science practitioner, Youssef has a strong passion for education. He is known for his leading data science and AI blog, newsletter, and eBooks on data science and machine learning.

Youssef is an applied scientist at Greenstep, focusing on building Generative AI features for Greenstep Products. He is also an AI applied researcher at Aalto University, working on AI agents and their applications. Before that, he worked as a researcher applying deep learning and computer vision techniques to medical images.