Time Series Forecasting in Python

Marco Peixeiro

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Time Series Forecasting in Python
Version 3

For more information on this and other Manning titles go to
manning.com

welcome

Thank you for purchasing the MEAP for *Time Series Forecasting in Python*.

This book is meant for the data scientist who already knows how to work with data and has solved regression problems, but who is unsure how to deal with time series. Thus, we assume you are comfortable using Python and libraries like numpy, pandas, and matplotlib.

While other resources teach time series using R, or a mix of Python and R, this book is 100% in Python. Python is the most popular programming language in data science, and it is time that we dedicate an entire book to time series forecasting using Python only.

Throughout the book, you will gain an intuition about time series properties and apply both simple and more complex forecasting models that take into account seasonal patterns and external variables. Plus, we study large datasets with high dimensionality using deep learning. Along the way, you will learn how to automate the process of time series forecasting.

To get the most out of this book, I highly suggest that you code along. Each chapter presents a real-life case study, followed by a theoretical concept, and its application in Python to solve the case. That way, you get both the theoretical foundation and technical knowledge to solve any time series problem. Plus, each section ends with a capstone project to help you consolidate your learning. This hands-on approach is sure to take you from beginner to well-rounded data scientist in time series.

I hope you enjoy reading the book as much as I enjoyed writing it! Please, leave your comments and feedback in the liveBook Discussion forum so that we can make this book the go-to resource for time series forecasting in Python.

Marco Peixeiro

brief contents

1 Understanding time series forecasting

This chapter covers

- Introducing time series
- Understanding the three main components of a time series
- Knowing what steps are necessary for a successful forecasting project
- How forecasting time series is different from other regression tasks

Time series exist in a variety of fields from meteorology to finance, econometrics, and marketing. With an added capacity of recording data and analyzing it, we can now study time series to analyze industrial processes or track business metrics, such as sales or engagement. Also, with large amounts of data being available, this represents a new opportunity for data scientists to apply their expertise to techniques for time series forecasting.

You might have come across other courses, books, or articles on time series that implement their solution in R. This programming language was specifically made for statistical computing. Many forecasting techniques make use of statistical models, as we will learn in chapter 3 and onwards. Thus, a lot of work was done to develop packages to make time series analysis and forecasting seamless using R. On the other hand, most data scientists are required to be proficient with Python, as it is the most widespread language in the field of machine learning. In recent years, the community and large companies have developed powerful libraries that leverage Python to perform statistical computing, machine learning tasks, develop websites, and much more. While Python is far from being a perfect programming language, its versatility is a strong benefit to its users, as we can develop models, perform statistical tests, and possibly serve our model through an API or develop a web interface, all while using the same programming language. Therefore, this book will

show you how to implement both statistical learning techniques and machine learning techniques for time series forecasting using only Python.

This book will focus entirely on time series forecasting. We will first learn how to make simple forecasts that will serve as benchmarks for more complex models. Then, we will use two statistical learning techniques, the moving average model, and the autoregressive model, to make forecasts. These will serve as the foundation to the more complex modeling techniques we will learn that will allow us to account for non-stationarity, seasonality effects, and impact of exogenous variables. Afterwards, we switch from statistical learning techniques to deep learning methods, in order to forecast very large time series with a high dimensionality, a scenario in which statistical learning often does not perform as well as its deep learning counterpart.

For now, this chapter will examine the basic concepts of time series forecasting. We start by defining time series so that we can recognize one. Then, we will move on and learn the purpose of time series forecasting. Finally, you will learn why forecasting a time series is different from other regression problems, and thus deserves its own book on the subject.

1.1 Introducing time series

The first step in understanding and performing time series forecasting is learning what a time series is. In short, a time series is simply a set of data points ordered in time. Furthermore, the data is often equally spaced in time, meaning that equal intervals separate each data point. In simpler terms, the data can be recorded at every hour, every minute, or it could be averaged over every month, or year. Some typical examples of time series include the closing value of a particular stock, a household's electricity consumption, or the temperature outside.

Time series

A time series is a set of data points ordered in time.

The data is equally spaced in time, meaning that it was recorded at every hour, or minute, or month, or quarter. Typical examples of time series include the closing value of a stock, a household's electricity consumption, or the temperature outside.

Let's consider the following dataset representing the quarterly earnings per share in U.S. dollars of Johnson & Johnson stock from 1960 to 1980. We will use this dataset often throughout this book, as it has many interesting properties that will help you learn advanced techniques for more complex forecasting problems.

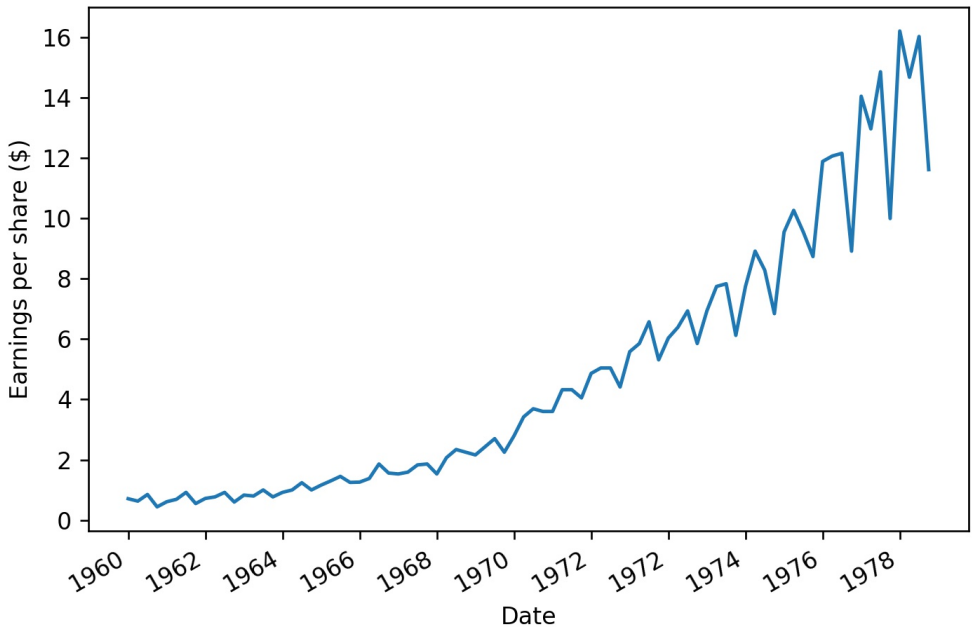


Figure 1.1 Quarterly earnings of Johnson & Johnson in USD from 1960 to 1980 showing a positive trend and a cyclical behavior

As you can see, figure 1.1 clearly represents a time series. The data is indexed by time, as illustrated on the horizontal axis. Also, the data is equally spaced in time since it was recorded at the end of every quarter for each year.

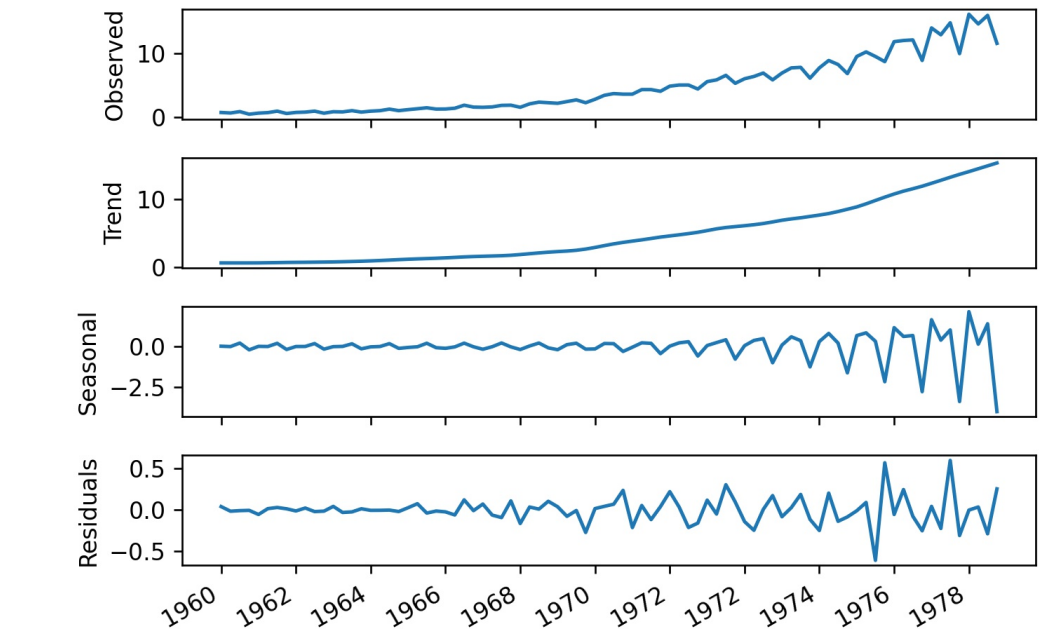
Also, we can see that the data has a trend, since the values are increasing over time. We see the data going up and down over the course of a year, and the pattern repeats itself every year. Those are known as components of a time series.

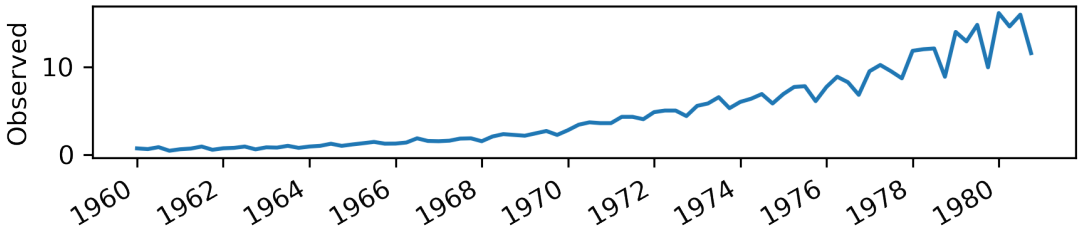
1.1.1 Components of a time series

We can further our understanding of time series by looking at its three components: a trend, a seasonal component, and residuals. In fact, all time series can be decomposed into these three elements.

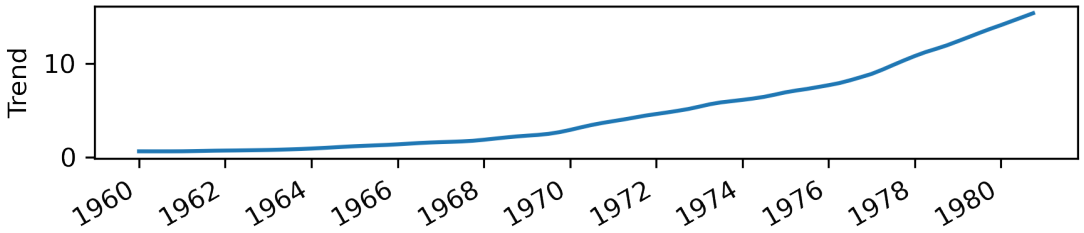
Visualizing the components of a time series is known as decomposition. Decomposition is defined as a statistical task that separates a time series into its different components. We can visualize each individual component which helps us identify the trend and seasonal pattern in our data, which is not always straightforward just by looking at our dataset.

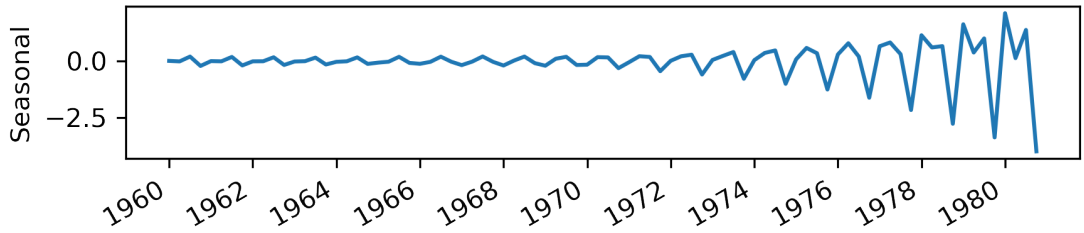
Let's take a closer look at the decomposition of Johnson & Johnson quarterly earnings per share in Figure 1.2. You can see how the *Observed* data was split into *Trend*, *Seasonal* and *Residuals*. Let's study each piece of the graph in more detail.





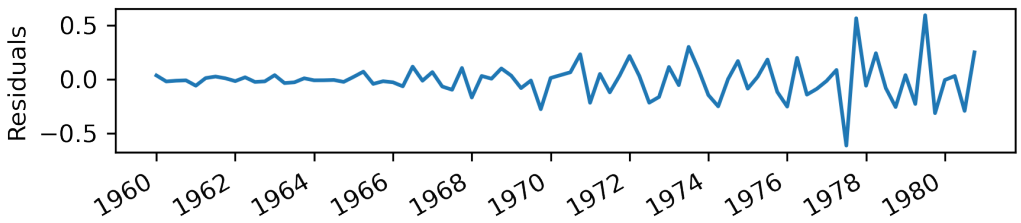
First, the top graph labeled as *Observed* simply shows the time series as it was recorded. The y-axis displays the value of the quarterly earnings per share for Johnson & Johnson in U.S Dollars, while the x-axis represents time. It is basically a recreation of Figure 1.1, and it shows the result of combining *Trend*, *Seasonal*, and *Residuals* from Figure 1.2.





Next, we see the seasonal component. The seasonal component captures the seasonal variation, which is a cycle that occurs over a fixed period of time. We can see that over the course of a year, or four quarters, the earnings per share start low, increase, and decrease again at the end of the year.

Notice how the y-axis shows negative values. Does it mean that the earnings per share are negative? Clearly, that cannot be since our dataset strictly has positive values. Therefore, we can say that the seasonal component shows how we deviate from the trend. Sometimes, we have a positive deviation, and we get a peak in *Observed*. Other times, we have a negative deviation, and we see a trough in *Observed*.



Finally, the last row shows the residuals, which is what cannot be explained by either the trend or the seasonal component. We can think of the residuals as adding *Trend* and *Seasonal* together and comparing the value at each point in time to *Observed*. For certain points, we might get the exact same value as in *Observed*, in which case the residual will be zero. In other cases, the value is different than the one in *Observed*, and so *Residuals* show what value must be added to *Trend* and *Seasonal* in order to adjust the result and get the same value as in *Observed*. Usually, residuals correspond to random errors, also termed white noise as we will cover in chapter 3. They represent information that we cannot model or predict, since it is completely random.

Time series decomposition

Time series decomposition is a process by which we separate a time series into its components: trend, seasonality, and residuals.

The trend represents the slow-moving changes in a time series. It is responsible for making the series gradually increase or decrease over time.

The seasonality component represents the seasonal pattern in the series. The cycles occur repeatedly over a fixed period of time.

The residuals represent the behavior that cannot be explained by the trend and seasonality components. They correspond to random errors, also termed white noise.

Already, we can intuitively see how each component affects our work when forecasting. If time series exposes a certain trend, then we expect it to continue in the future. Similarly, if we observe a strong seasonality effect, this is likely going to continue, and our forecasts must reflect that. Later in the book, we will see how to account for these components and include them in our models to forecast more complex time series.

1.2 Bird's-eye view of time series forecasting

Forecasting is the prediction of the future using historical data and knowledge of future events that might affect our forecasts. This definition is full of promises and, as data

scientists, we are often very eager to start forecasting by using our scientific knowledge in order to showcase an incredible model with a near-perfect forecast accuracy.

However, there are important steps that must be covered before reaching the point of forecasting. Figure 1.3 is a simplified diagram of what a complete forecasting project may look like in a professional setting. Note that these steps are not universal, and they may or may not be followed depending on the organization and its maturity. They are nonetheless essential to ensure a good cohesion between the data team and the business team, hence providing business values and avoiding friction and frustrations between both teams.

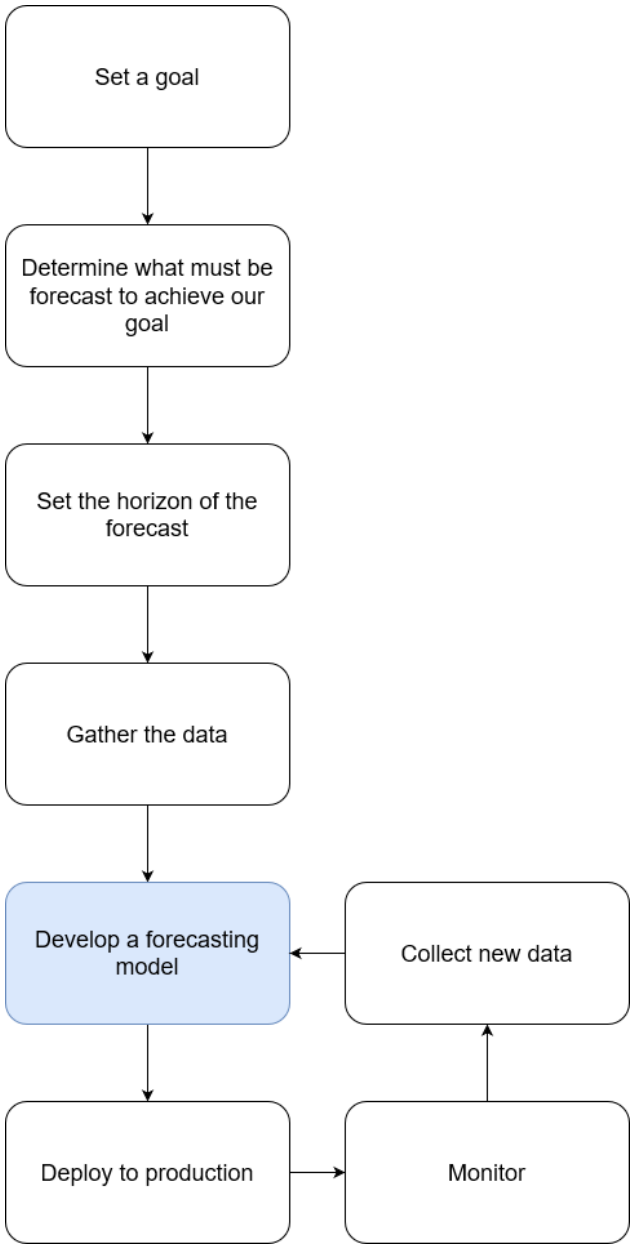


Figure 1.3 Forecasting project roadmap. The first step is naturally to set a goal that justifies the need of forecasting. Then, we must determine what needs to be forecast in order to achieve that goal. Then, we set the horizon of the forecast. Once done, we can gather the data and develop a forecasting model. Then, the model is deployed to production, its performance is monitored, and new data is collected in order to retrain the forecasting model and make sure it is still relevant.

Let's dive into a scenario to cover each step of a forecasting project roadmap in detail. Imagine you are planning a one-week camping trip one month from now, and you want to know which sleeping bag to bring with you, so you can sleep comfortably at night.

SET A GOAL

The very first step in any project roadmap is to set a goal. Here, it is explicit from the scenario: you want to know which sleeping bag to bring to sleep comfortably at night. If the nights will be cold, then a warm sleeping is the best choice. Of course, if nights are expected to be warm, then a light sleeping bag would be the better option.

DETERMINE WHAT MUST BE FORECAST TO ACHIEVE OUR GOAL

Then, we move to determining what must be forecast in order for us to decide which sleeping bag to bring. In this case, we need to predict the temperature at night. To simplify things, let's consider that predicting the minimum temperature is sufficient to make a decision, and that the minimum temperature occurs at night.

SET THE HORIZON OF THE FORECAST

Now, we can set the horizon of our forecast. In this case, your camping trip is one month from now, and it will last for one week. Therefore, we have a horizon of one week since we are only interested in predicting the minimum temperature during the camping trip.

GATHER THE DATA

We can now start gathering our data. For example, we could collect historical daily minimum temperature data. We could also gather data on possible factors that influence temperature, such as humidity and wind speed.

This is when the question of how much data is enough data arises. Ideally, we would collect more than one year of data. That way, we can determine if there is a yearly seasonal pattern or a trend. In the case of temperature, we can of course expect some seasonal pattern over the year since different seasons bring different minimum temperatures.

However, one year of data is not the ultimate answer to how much data is sufficient. It highly depends on the frequency of the forecasts. In our case, we will have daily forecasts, so one year of data should be enough.

If we wanted to do hourly forecasts, then a few months of training data would be enough, as it would contain a lot of data points. In the case where we would do monthly or yearly forecasts, then we would need a much larger historical period to have enough data points to train with.

In the end, there is no clear answer to the quantity of data required to train a model. It is part of the experimentation process of building a model, assessing its performance, and testing if more data improves the model's performance.

DEVELOP A FORECASTING MODEL

With our historical data in hand, we are ready to develop a forecasting model. This part of the project roadmap is the focus of this entire book. This is when you get to study the data and determine if there is a trend or a seasonal pattern.

If we observe seasonality, then a SARIMA model would be relevant because this model uses seasonal effects to produce forecasts. If we have information on wind speed and humidity, then we can take that into account using the SARIMAX model because we can feed it with information coming from exogenous variables, such as wind speed and humidity.

In the case that we managed to collect a large amount of data, such as the daily minimum temperature of the last twenty years, then we could use neural networks in order to leverage this very large amount of training data. This is because, unlike statistical learning methods, deep learning tends to produce better models as more data is used for training

Whichever model you develop, you will use part of the training data as a test set to evaluate your model's performance. The test set will always be the most recent data points and it must be representative of the forecasting horizon.

In this case, since our horizon is one week, we remove the last seven data points from our training set to place them in a test set. Then, when each model is trained, we produce one-week forecasts and compare the results to the test set. The model's performance is assessed by computing an error metric, such as the mean squared error (MSE) for example. This is a way to evaluate how far our predictions are from the real values. The model with the lowest MSE is our best performing model and it is the one that will move on to the next step.

DEPLOY TO PRODUCTION

Once we have our champion model, we must deploy it to production. This means that our model can take in data and return a prediction for the minimum daily temperature for the next seven days. There are many ways to deploy a model to production, and this can be a subject for an entire book. Your model could be served as an API, or integrated in a web application, or you could define your own Excel function to run your model. Ultimately, your model is deployed when we can feed in data and have forecasts back without any manual manipulation of the data. At this point, your model can then be monitored.

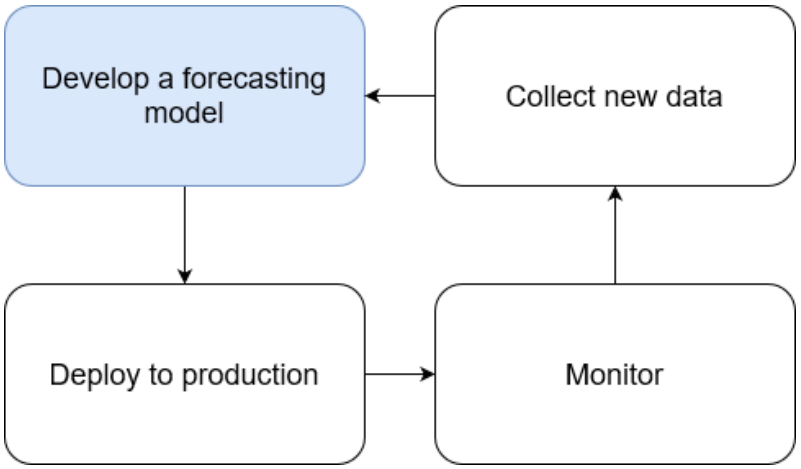
MONITOR

Since the camping trip is one month from now, we get to see how well our model performs. Every day, we will compare our model's forecast to the actual minimum temperature recorded for the day. This allows us to determine the quality of the model's forecasts.

We can also look for unexpected events. For example, a heat wave can arise, degrading the quality of your model's forecasts. Closely monitoring your model and current events allows you to determine if it is a temporary situation, or if it will last for next two months, in which case it will impact your decision for the camping trip.

COLLECT NEW DATA

From monitoring your model, you necessarily collect new data, as you compare the model's forecasts to the observed minimum temperature for the day. This new, more recent data can then be used in retraining our model. That way, we have up-to-date data to forecast the minimum temperature for the next seven days.



This cycle is then repeated over the next month until we reach the day of the camping trip. By that point, you will have made many forecasts, assessed their quality against newly observed data, and retrained your model with new daily minimum temperature as you recorded them. That way, you make sure that your model is still performant and uses relevant data to forecast the temperature for your camping trip. Finally, based on your model's predictions, you will then decide which sleeping bag to bring with you.

1.3 How time series forecasting is different from other regression tasks

You probably have encountered previous regression tasks where you must predict some continuous target given a certain set of features. At first glance, time series forecasting seems like a typical regression problem: we have some historical data, and we wish to build a mathematical expression that will express future values as a function of past values. However, there are some key differences between time series forecasting and regression for time-independent scenarios that deserve to be addressed before learning our very first forecasting technique.

1.3.1 Time series have an order

The first concept to keep in mind is that time series have an order, and we cannot change that order when modeling.

In time series forecasting, we express the future values as a function of past values. Therefore, we must keep the data in order, such as to not violate this relationship.

Also, it makes sense to keep the data in order because your model can only use the information from the past up until the present, since it will not know what will be observed in the future. Let's recall your camping trip that we covered with a project roadmap. If you want to predict the temperature for Tuesday, you cannot possibly use the information from Wednesday, since it is in the future from the model's point of view. You would only be able to

use the data from Monday and before. That is why the order of the data must remain the same throughout the modeling process.

Other regression tasks in machine learning often do not have an order. For example, if you are tasked to predict the revenue based on ad spend, it does not matter when a certain amount was spent on ads. Instead, you simply want to relate the amount of ad spend to the revenue. In fact, you might even randomly shuffle the data to make your model more robust. Here, the regression task is to simply derive a function such that given an amount on ad spend, an estimate of revenue is returned.

On the other hand, time series are indexed by time, and that order must be kept. Otherwise, you are training your model with future information that it will not have at prediction time. The resulting model will therefore not be reliable and will most probably perform poorly when you will make future forecasts.

1.3.2 Time series sometimes do not have features

It is possible to forecast time series without the use of other features than the time series itself.

As data scientists, we are used to having datasets with many columns, each representing a potential predictor for our target. For example, consider the task of predicting revenue based on ad spend, where the revenue is the target variable. As features, we could have the amount spent on Google ads, Facebook ads, and television ads. Using these three features, we would build a regression model to estimate revenue.

However, with time series, it is quite common to be given a simple dataset with a time column, and a value at that point in time. Without any other features, we must learn ways of using past values of the time series to forecast future values. This is when the moving average model (chapter 4) or autoregressive model (chapter 5) come into play, as they are ways to express future value as a function of past values. These models are foundational to the more complex models that then allow you to consider seasonal patterns and trends in time series. Starting from chapter 6, we will gradually build upon those basic models in order to forecast more complex time series.

1.4 Next steps

This book will cover different forecasting techniques in detail. We start with some very basic methods, such as the moving average model and autoregressive model, and we gradually account for more factors in order to forecast time series with trends and seasonal patterns using the ARIMA, SARIMA and SARIMAX models. We will also work with time series with a high dimensionality, which will require us to use deep learning techniques for sequential data. Therefore, we will have to build neural networks using CNN (convolutional neural network) and LSTM (long short-term memory). Finally, we will learn how to automate the work of forecasting time series. As mentioned, all implementations will be done in Python throughout the book.

Now that we learned what a time series is and how forecasting will be different than traditional regression tasks you might have seen up until now, we are ready to move on and

start forecasting. However, since it will be our first attempt at forecasting, we will focus on naïve methods that will serve as baseline models.

1.5 Summary

- A time series is a set of data points ordered in time.
- Examples of time series are the closing price of a stock or the temperature outside.
- Time series can be decomposed into three components: a trend, a seasonal component, and residuals.
- It is important to have a goal when forecasting and to monitor the model once deployed. This will ensure the success and longevity of the project.
- Never change the order of a time series when modeling. Shuffling the data is not allowed.

2

A naïve prediction of the future

This chapter covers

- Defining a baseline model
- Setting a baseline using the mean
- Setting a baseline using the mean previous window of time
- Setting a baseline using the previous time step
- Implementing the naïve seasonal forecast

In chapter 1, we covered what time series are and how forecasting a time series is different from a traditional regression task. Also, we learned the necessary steps to build a successful forecasting project, from defining a goal to building model, deploying it, and updating it as new data is collected. Now, we are ready to start forecasting a time series.

We will first learn how to make a naïve prediction of the future, which will serve as a baseline. The baseline model is a trivial solution that uses heuristics, or simple statistics, to compute a forecast. Developing a baseline model is not always an exact science. It will often require some intuition that we gain by visualizing the data and detecting patterns that can be used to make predictions. In any modeling project, it is important to have a baseline, as we will use it to compare the performance of the more complex models we will build down the road. The only way to know that a model is good, or performant, is to compare it to a baseline.

In this chapter, let's imagine that we wish to predict the quarterly earnings per share (EPS) of Johnson & Johnson. We can look at the dataset in figure 2.1, which is identical to what we introduced in chapter 1. Specifically, we will want to use the data from 1960 to the end of 1979 in order to predict the earnings per share for the four quarters of 1980. The forecasting period is illustrated by the gray zone in figure 2.1.

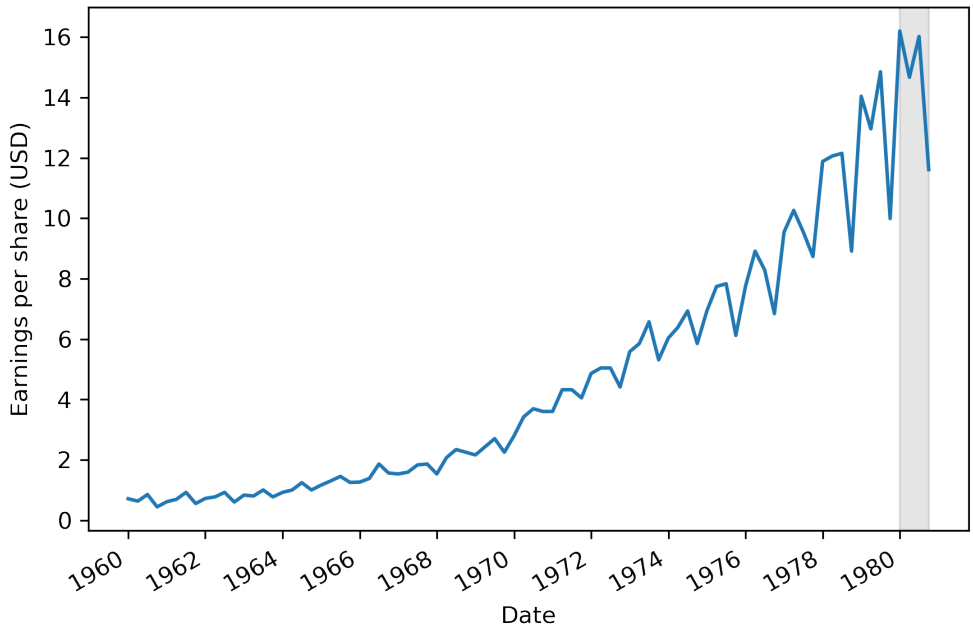


Figure 2.1 Quarterly earnings per share of Johnson & Johnson in U.S Dollars (USD) between 1960 and 1980. We will use the data from 1960 to the last quarter of 1979 to build a baseline model that will forecast the earnings per share for the quarters of 1980 (as illustrated by the gray area).

We can see in figure 2.1 that our data has a trend, since it is increasing over time. Also, we have a seasonal pattern since over the course of a year, or four quarters, as we observe peaks and troughs repeatedly. This means that we have seasonality. Recall that we identified each of these components when we decomposed our time series in chapter 1. We can look at each component in figure 2.2.

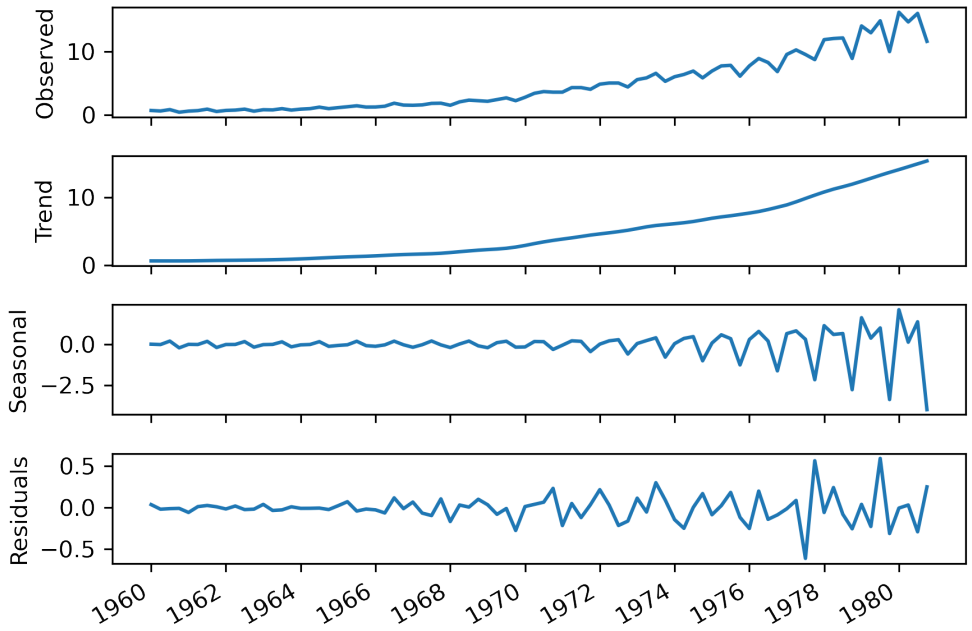


Figure 2.2 Decomposition of quarterly earnings of Johnson & Johnson from 1960 to 1980.

We will study some of these components in detail later in the chapter, as they will help us gain some intuition about the behavior of the data, which in turn will help us develop a good baseline model.

We first define what a baseline model is and then we will develop four different baselines to forecast the quarterly earnings per share of Johnson & Johnson. This is the time where we finally get our hands dirty with Python and time series forecasting.

2.1 Defining a baseline model

A baseline model is a trivial solution to our problem. It often uses heuristics, or simple statistics, to generate predictions. The baseline model is the simplest solution you can think of—it should not require any training and the cost of implementation should be very low.

> **CAN YOU THINK OF BASELINE FOR OUR PROJECT?** Knowing that we want to forecast the earnings per share for Johnson & Johnson, what is the most basic, most naïve, forecast you can make?

In the context of time series, one simple statistic we can use to build a baseline is the arithmetic mean. We can simply compute the mean of the values over a certain period and assume that future values will be equal to that mean. In the context of predicting the EPS for Johnson & Johnson, this is like saying:

> The average EPS between 1960 and 1979 was $4.31. Therefore, I expect the EPS over the next four quarters of 1980 to be equal to $4.31.

Another possible baseline is to naively forecast the last recorded data point. In our context, it would be like saying:

> If the earnings per share are $0.71 for this quarter, then the earnings per share will also be $0.71 for next quarter.

Or, if we see a cyclical pattern in our data, we can simply repeat that pattern into the future. Staying in the context of Johnson & Johnson, this is like saying:

> If the earnings per share are $14.04 for the first quarter of 1979, then the earnings per share for the first quarter of 1980 will also be $14.04.

You can see how all three possible baselines mentioned above rely on simple statistics, heuristics, and patterns observed in our dataset.

Baseline model

A baseline model is a trivial solution to our forecasting problem.

It relies on heuristics or simple statistics. It usually the simplest solution; it does not require model fitting and it is easy to implement.

Now, you might wonder if those baseline models are any good. How well can those simple methods forecast the future? We answer this question by forecasting for the year of 1980 and testing our forecasts against the observed data in 1980. This is called *out-of-sample* forecasting, because we are making predictions for a period that was not taken into account when the model was developed. That way, we can measure the performance of our models and see how they would perform when we forecast beyond the data we have, which in this case is 1981 and later.

In the next sections, we will learn how to develop different baselines we discussed above to predict the quarterly earnings per share of Johnson & Johnson.

2.2 Forecasting the historical mean

As mentioned at the beginning of the chapter, we are going to work with the quarterly earnings per share in US Dollars (USD) of Johnson & Johnson from 1960 to 1980. Our goal is to use the data from 1960 to the end of 1979 to predict the last quarters of 1980. The first baseline we cover uses the historical mean, which is the arithmetic mean of past values. Its implementation is straightforward: calculate the mean of our training set, and it will be our prediction for the four quarters of 1980. Let's see how we can implement it using Python.

Our first step is to load the dataset. To do so, we are going to use the `pandas` library and load the dataset into a `DataFrame` using the method `read_csv`. You can either download on

your local machine and pass the path to the file to the `read_csv` method, or simply type in the URL where the CSV file is hosted on Github. In this case, we are going to work with the URL.

```
import pandas as pd

df =
       pd.read_csv('https://raw.githubusercontent.com/marcopeix/AppliedTimeSeriesAnalysisWi
       thPython/main/data/jj.csv')
```

A `DataFrame` is the most used data structure in `pandas`. It is a 2-dimensional labeled data structure with columns that can hold different types of data such as strings, integers, floats, or dates.

Next, we will split the data into a training set and a test set. Given that our horizon is one year, our training set will start in 1960 and will go all the way to the end of 1979. We will save the data collected in 1980 for our test set. We can think of a `DataFrame` as a table or a spreadsheet with column names and row indices.

With our dataset in a DataFrame, we can display the first five entries by running:

```
df.head()
```

	date	data
0	1960-01-01	0.71
1	1960-04-01	0.63
2	1960-07-02	0.85
3	1960-10-01	0.44
4	1961-01-01	0.61

Figure 2.3 The first five entries of quarterly earnings per share for Johnson & Johnson dataset. Notice how our `DataFrame` has two columns: date and data. We also see that our `DataFrame` has row indices starting at 0.

From figure 2.3, we can better understand what type of data our `DataFrame` is holding. We have the *date* column which specifies the end of each quarter at which the earnings per share are calculated. Then, the *data* column holds the value of the earnings per share in U.S dollars (USD).

We can optionally display the last five entries of our dataset and obtain the output of figure 2.4.

```
df.tail()
```

	date	data
79	1979-10-01	9.99
80	1980-01-01	16.20
81	1980-04-01	14.67
82	1980-07-02	16.02
83	1980-10-01	11.61

Figure 2.4 The last five entries of our dataset. Here, we can see the four quarters of 1980 that we will try to predict using different baseline models. We will compare our forecasts to the observed data in 1980 to evaluate the performance of each baseline.

In figure 2.4, we see the four quarters of 1980 which is what we will be trying to forecast using our baseline model. We will compare our forecasts to the values from the data column for the four quarters of 1980 to evaluate the performance of our baseline. The closer our forecast is to the observed value, the better.

The final step before developing our baseline model is to split the dataset into a *train* and *test* set. As mentioned earlier, the train set will consist of the data from 1960 to the end of 1979, and the test set will consist of the four quarters of 1980.

The train set will be considered as the only information we have available to develop our model. Once the model is built, we will forecast the next four timesteps which will correspond to the four quarters of 1980 in our test set. That way, we can compare our forecasts to the observed data and evaluate the performance of our baseline.

To make the split, we specify that our train set will contain all the data held in `df` except the last four entries. Then, the test set will be composed of only the last four entries. This exactly what the next code block does:

```
train = df[:-4]
test = df[-4:]
```

Now we are ready to implement our baseline. We will first use the arithmetic mean of the entire train set. To compute the mean, we use the `numpy` library, as it is a very fast package for scientific computing in Python that plays really well with `DataFrames`.

```
import numpy as np

historical_mean = np.mean(train.data)       #A

print(historical_mean)
```

#A Compute the arithmetic mean of the *data* column in the train set.

In the code block above, we first import the `numpy` library and then we compute the average of the EPS over the entire train set and print it out on the screen. This gives you a value 4.31 USD. This means that from 1960 to the end of 1979, the EPS of Johnson & Johnson is on average 4.31 USD.

Now, we will naively forecast this value for each quarter of 1980. To do so, we will initialize and empty `DataFrame` to hold our predictions. We specify that the *date* column should be equal to the dates in the test set, since we are forecasting for the same period. Then, we specify a *pred* column to hold our predicted values for each quarter:

```
pred_hist_mean = pd.DataFrame()    #A
pred_hist_mean['date'] = test.date
pred_hist_mean['pred'] = historical_mean

pred_hist_mean    #B
```

#A Initialize an empty `DataFrame`.
#B Display the `pred_hist_mean DataFrame`

$$MAPE = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{A_i - F_i}{A_i}\right|$$

Equation 2.1

In equation 2.1, A_i is the actual value at point i in time, and F_i is the forecast value at a point i in time. Then, n is simply the number of forecasts. In our case, because we are forecasting the four quarters of 1980, $n = 4$. Inside the summation, we see that the forecast value is subtracted from the actual value, and that result is divided by the actual value, which gives us the percentage error. Then, we take the absolute value of the percentage error. This operation is repeated for each n points in time and are added together. Finally, we divide the sum by the number of n points in time which effectively gives us the mean absolute percentage error.

Let's implement this function in Python. We define the function `mape` that takes in two vectors: `y_true` for the actual values observed in the test set, and `y_pred` for the forecast values. In this case, because `numpy` allows us to vectorize the function, we will not need a

loop to sum all values. We can simply subtract the `y_pred` vector from the `y_true` vector and divide by `y_true` to get the percentage error. Then, we can take the absolute value. After, we take the mean of the result, which will take care of summing up each value in the vector and dividing by the number of predictions. Finally, we multiply the result by 100 in order for the output to be expressed as a percentage instead of a decimal number:

```
def mape(y_true, y_pred):
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100
```

Now, we can calculate the MAPE of our baseline. Here, our actual values are in the *data* column of `test`, so it will be the first parameter passed to the `mape` function. Our forecasts are in the *pred* column of `pred_hist_mean`, and it will be our second parameter for the function:

```
mape_hist_mean = mape(test.data, pred_hist_mean.pred)

print(mape_hist_mean)
```

Running the function gives a MAPE of 70.00%. This means that our baseline deviates by 70% from the observed quarterly EPS of Johnson & Johnson in 1980.

Let's visualize our forecasts to better understand our MAPE of 70%.

Listing 2.1 Visualizing our forecasts

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

ax.plot(train.date, train.data, 'g-.', label='Train')
ax.plot(test.date, test.data, 'b-', label='Test')
ax.plot(pred_hist_mean.date, pred_hist_mean.pred, 'r--', label='Predicted')
ax.set_xlabel('Date')
ax.set_ylabel('Earnings per share (USD)')
ax.axvspan(80, 83, color='#808080', alpha=0.2)
ax.legend(loc=2)

plt.xticks(np.arange(0, 81, 8), [1960, 1962, 1964, 1966, 1968, 1970, 1972, 1974, 1976,
       1978, 1980])

fig.autofmt_xdate()
plt.tight_layout()
```

In listing 2.1, we use the `matplotlib` library, which is the most popular library for generating visualizations in Python, to generate a graph showing the training data, the forecast horizon, the observed values of the test set and the predictions for each quarter of 1980.

First, we initialize a `figure` and an `ax` object. A figure can contain many `ax` objects, which allows us to create a figure with two, three, or more plots. In this case, we are creating a figure with a single plot, so we only need one `ax`.

Second, we will plot on the `ax` object our data. We plot the train data using a green dashed and dotted line and give this curve a label of *Train*. The label will later be useful to generate a legend for our graph. We then plot the test data and use a blue continuous line,

with a label of *Test*. Finally, we plot our predictions using a red dashed line with a label of *Predicted*.

Third, we label our x-axis and y-axis. We then draw a rectangular area to illustrate the forecast horizon. Since our forecast horizon is the four quarters of 1980, the area should start at index 80 and end at index 83, spanning the entire year of 1980. Remember that we obtained the indices of the last quarter of 1980 by running `df.tail()`, which resulted in figure 2.5.

	date	data
79	1979-10-01	9.99
80	1980-01-01	16.20
81	1980-04-01	14.67
82	1980-07-02	16.02
83	1980-10-01	11.61

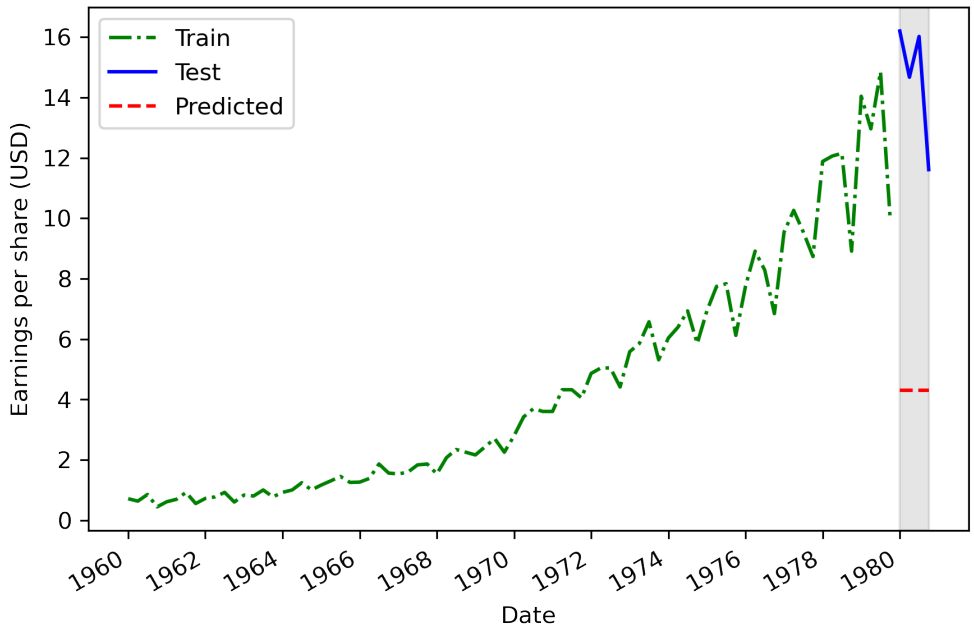
Figure 2.5 The last five entries of our dataset.

We give this area a grey color and specify the opacity using the `alpha` parameter. When `alpha` is 1, the shape is completely opaque; if `alpha` is 0 it is completely transparent. In our case, we use an opacity of 20% or 0.2.

Then, we specify the labels for the ticks on the x-axis. Otherwise, we would see the data for each quarter of the dataset, which create a crowded x-axis with unreadable labels. Instead, we will display the year every two years. To do so, we generate an array specifying the index at which the label must appear. That's what `np.arange(0, 81, 8)` does for us: we generate an array starting at 0, finishing at 80, because the end index (81) is not included, with steps of 8, because there are 8 quarters in 2 years. This will effectively generate the following array: $[0, 8, 16, \ldots 72, 80]$. Then, we specify an array containing the labels at each index, so it must start with 1960 and end with 1980, just like our dataset.

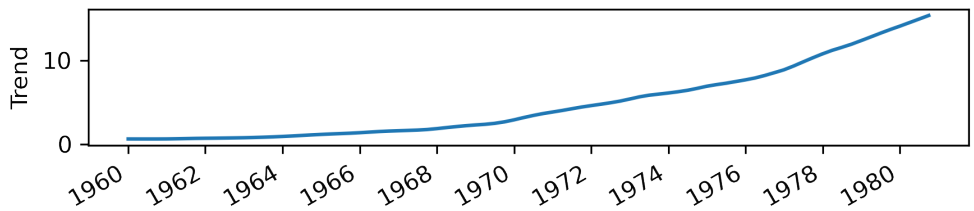
Finally, we use `fig.automft_xdate()` to automatically format the tick labels on the x-axis. It will slightly rotate them and make sure that it is legible. We bring the final touch up with `plt.tight_layout()` in order to remove excess white space around the figure.

Once the steps above are completed, we successfully generate figure 2.5.



Clearly, this baseline did not yield accurate predictions, since the *Predicted* line is very far off from the *Test* line. Now, we know that our forecasts are, on average, 70% below the actual EPS for each quarter in 1980. Whereas the EPS in 1980 was consistently above 10$, we predicted only 4.31$ for each quarter.

Still, what can we learn from it? Looking at our training set, we can see a positive trend, as the earnings per share are increasing over time. This is further supported by the trend component coming from the decomposition of our dataset.



As you we can see, not only do we have a trend, but the trend is not constant between 1960 and 1980, as we see it getting steeper. Therefore, it might be that the EPS observed in 1960

is not predictive of the EPS in 1980, because we have a positive trend and EPS values have kept increasing with time and have done so at a faster rate.

> **CAN YOU IMPROVE OUR BASELINE?** Before moving on to the next section, can you think of a way to improve our baseline while still using the mean? Do you think that taking the mean of a shorter and more recent period of time would help (from 1970 to 1979 for example)?

2.3 Forecasting last year's mean

The lesson learned from the previous baseline is that earlier values do not seem to be predictive of future values in the long-term because of the positive trend component of our dataset. Earlier values seem to be too small to be representative of the new level the EPS reaches towards the end of 1979 and onwards in 1980.

So, how about we consider the mean of the last year recorded on our training set to forecast the following year? This means that we will compute the average EPS in 1979 and forecast it for each quarter of 1980—that way, we take more recent values that have increased over time and should potentially be closer to what will be observed in 1980. For now, this is simply a hypothesis so let's implement this baseline and test it to see how it performs.

With our data already split into a test and training sets, we can calculate right away the mean of the last year recorded on the training set, which corresponds to the last four data points in 1979:

```
last_year_mean = np.mean(train.data[-4:])   #A

print(last_year_mean)
```

#A Compute the average EPS for the four quarters of 1979, which are the last 4 data points of the train set.

This gives us an average EPS of 12.96 USD. Therefore, we will predict that the Johnson & Johnson will have an EPS of 12.96$ for the four quarters of 1980. Using the same procedure that we used for the previous baseline, we will initialize a `Dataframe` called `pred_last_year_mean`, and assign a column containing the dates corresponding to the four quarters of 1980 present in the test set. Then, we will assign `last_year_mean` to the *pred* column:

```
pred_last_year_mean = pd.DataFrame()
pred_last_year_mean['date'] = test.date
pred_last_year_mean['pred'] = last_year_mean
```

Then, using the `mape` function that we defined earlier, we can evaluate the performance of our new baseline. Remember that the first parameter is the observed values, which are held in the test set. Then, we pass the predicted values which are in the *pred* column of `pred_last_year_mean`.

```
mape_last_year_mean = mape(test.data, pred_last_year_mean.pred)
print(mape_last_year_mean)
```

This gives us a MAPE of 15.60%. We can visualize our forecasts in figure 2.7.

CAN YOU RECREATE FIGURE 2.7? As an exercise, try to recreate figure 2.7 to visualize the forecasts using the mean of the quarters of 1979. The code should be identical to what we covered in section 2.1, only this time the predictions are in a different `DataFrame`.

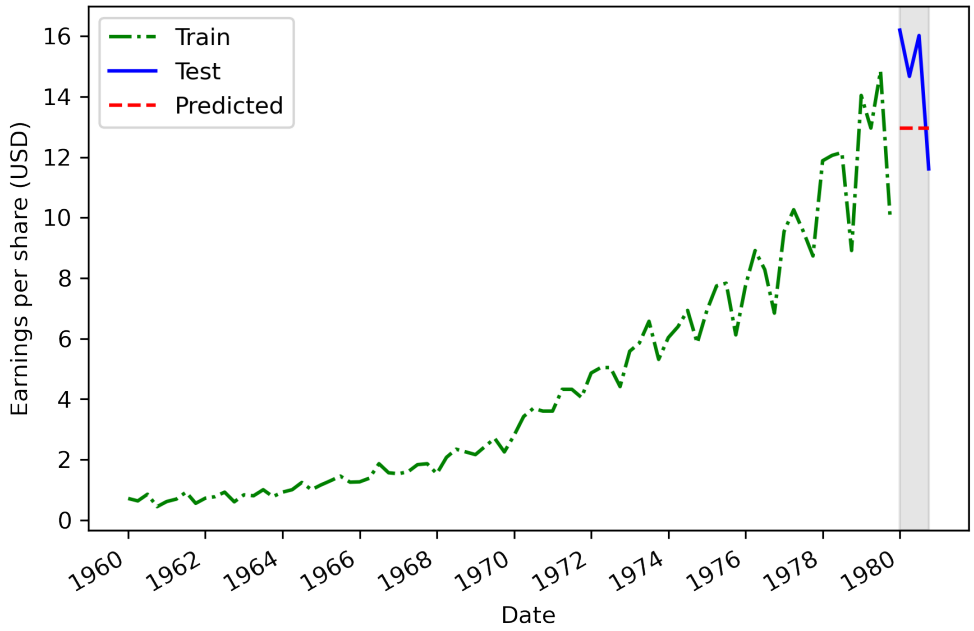


Figure 2.7 Predicting the mean of the last year recorded on the training set (1979) as a baseline model. We can see that the prediction is closer to actual values of the test set when compared to the previous baseline that we built in figure 2.5.

This new baseline is a clear improvement upon the previous one as we decreased the MAPE from 70% to 15.6%, even though its implementation is just as simple. This means that our forecasts deviate from the observed values by 15.6% on average. Using the last year's mean is a good step in the right direction, as we want to get a MAPE as close to 0% as possible, since it would translate into predictions that are closer to the actual values in our forecast horizon.

Therefore, we can learn from this baseline that future values likely depend on past values that are not too far back in history. This is a sign of autocorrelation, and we will dive deep into this subject in chapter 5. For now, let's see another baseline that we could develop for this situation.

2.4 Predicting using the last known value

Previously we used the mean over different periods to develop a baseline model. For now, the best baseline was the mean of the last recorded year of our training set since it yielded the lowest MAPE. We learned from that baseline that future values depend on past values, but not too far back in time. Indeed, predicting the mean EPS from 1960 to 1979 yield worse forecasts than predicting the mean EPS over 1979.

Therefore, we could suppose that using the last known value of the training set as a baseline model will give us even better forecasts, which would translate in a MAPE close to 0%. Let's test that hypothesis.

The first step is to extract the last known value of our training set, which corresponds to the earnings per share recorded for the last quarter of 1979:

```
last = train.data.iloc[-1]

print(last)
```

We retrieve the EPS recorded on the last quarter of 1979 and get a value of 9.99 USD. We will thus predict that Johnson & Johnson will have an EPS of 9.99 USD for the four quarters of 1980.

Again, we will create a `DataFrame` to contain our prediction for the test set. Here, we call it `pred_last`, and assign last to the *pred* column:

```
pred_last = pd.DataFrame()
pred_last['date'] = test.date
pred_last['pred'] = last
```

Then, using the same MAPE function that we defined earlier, we evaluate the performance of this new baseline model. Again, we pass to the function the actual values from the test set, and our prediction from the *pred* column of `pred_last`:

```
mape_last = mape(test.data, pred_last.pred)

print(mape_last)
```

This gives us a MAPE of 30.45%. We can visualize the forecasts in figure 2.8.

> **TRY TO PRODUCE FIGURE 2.8 ON YOUR OWN!** As a data scientist, it is important to convey our results in a way that is accessible to people who do not work in our domain. Thus, producing plots showing our forecasts is an important skill to develop.

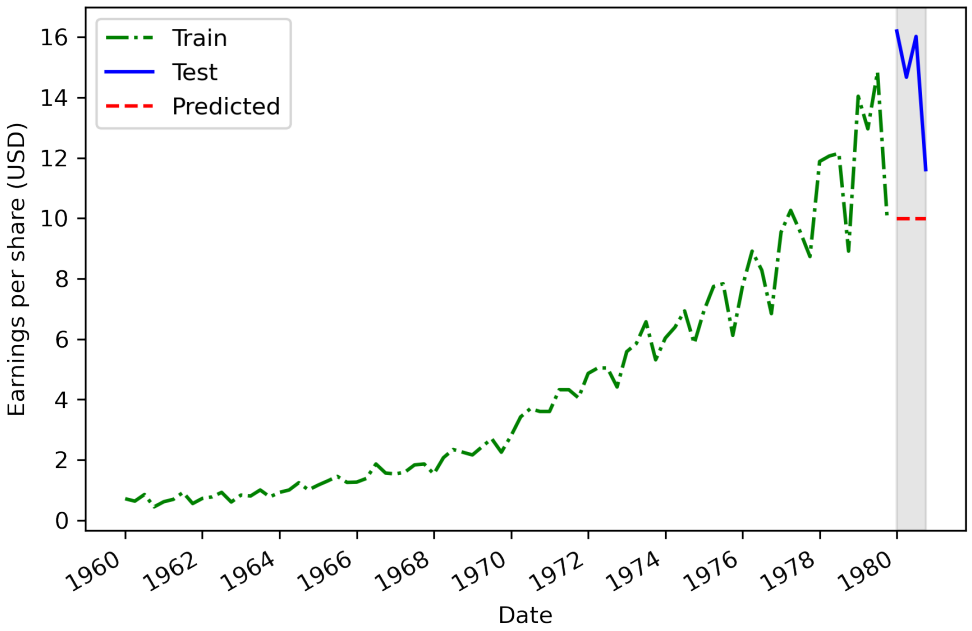


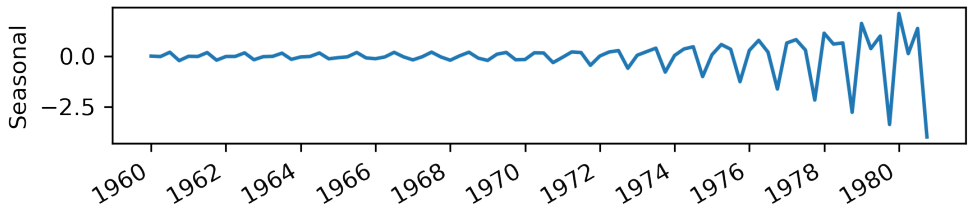
Figure 2.8 Predicting the last known value of the training set as a baseline model. We can see that this baseline is better than the first one, but less performant than our second baseline with a MAPE of 30.45%.

It seems that our new hypothesis did not improve upon the last baseline that we built since we have a MAPE of 30.45%, whereas we achieved a MAPE of 15.60% using the mean EPS over 1979. Therefore, these new forecasts are farther from the observed values in 1980.

This can be explained by the fact that the EPS displays a cyclical behavior, where it is high during the first three quarters, before falling at the last quarter. Using the last known value does not take the seasonality into account, so we need to use another naïve forecasting technique to see if we can produce a better baseline.

2.5 Implementing the naïve seasonal forecast

While we considered the trend component for the first two baselines in this chapter, we have not studied another important component from our dataset, which is the seasonal component.



Clearly, we can see cyclical patterns in our data and that is a piece of information that we could use to construct one last baseline: the naïve seasonal forecast.

The naïve seasonal forecast takes the last observed cycle and repeats it into the future. In our case, a full cycle occurs in four quarters. Therefore, we will take the EPS from the first quarter of 1979 and predict that value for the first quarter of 1980. Then, we take the EPS from the second quarter of 1979 and predict that value for the second quarter of 1980. The process is repeated for both the third and fourth quarters.

In Python, we implement this baseline by simply taking the last four values of the train set, which correspond to each quarter of 1979, and assigning them to the corresponding quarter in 1980. Here, we define `pred_naive_seasonal` to hold our predictions from the naïve seasonal forecast method:

```
pred_naive_seasonal = pd.DataFrame()
pred_naive_seasonal['date'] = test.date
pred_naive_seasonal['pred'] = train.data[-4:].values #A
```

#A Our predictions are the last four values of our train set, which correspond to each quarter of 1979.

Then, we calculate the MAPE the same way we did in the previous sections:

```
mape_naive_seasonal = mape(test.data, pred_naive_seasonal.pred)

print(mape_naive_seasonal)
```

This gives a MAPE 11.56%, which is the lowest MAPE from all baselines in this chapter. Figure 2.9 illustrates our forecast compared to the observed data in the test set. As an exercise, I strongly suggest that you try to recreate it on your own.

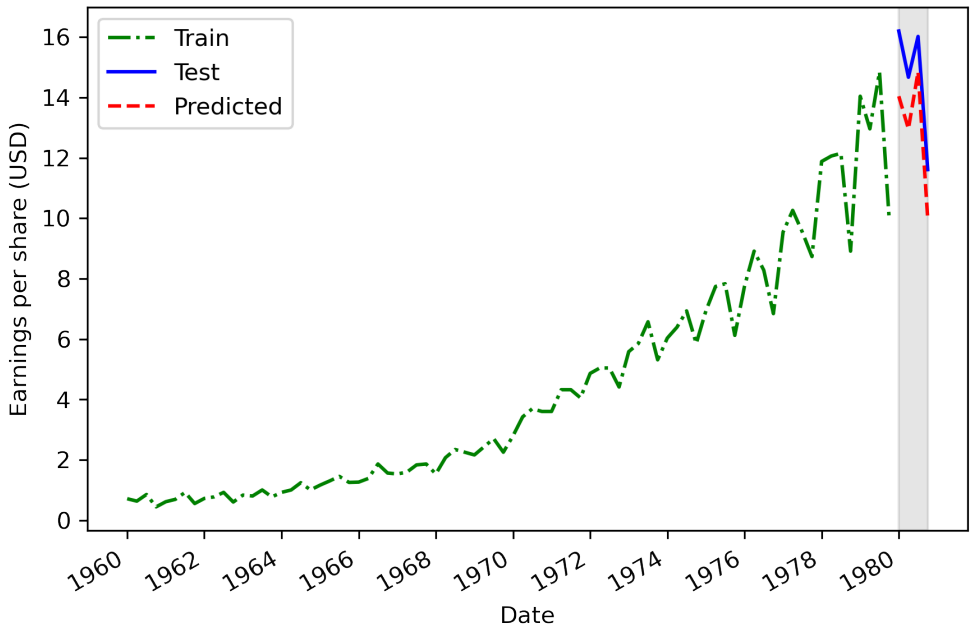


Figure 2.9 Result of the naïve seasonal forecast on the test set. As you can see, this forecast is more similar to the data observed in the test set, which resulted in the lowest MAPE. Clearly, the seasonality of this dataset has an impact on future values, and it must be considered when forecasting.

As you can see, using a naïve seasonal forecast resulted in the lowest MAPE of all baselines that we built throughout this chapter. This means that seasonality has a significant impact on future values, since repeating the last season into the future yields fairly accurate forecasts. Intuitively, this makes sense as well as we can clearly observe a cyclical pattern being repeated every year in figure 2.9. Seasonal effects will have to be considered when we develop a more complex forecasting model for this problem. We will explain in detail how to account for them in chapter 8.

2.6 Next steps

In this chapter, we developed four different baselines for our forecasting project. We used the arithmetic mean of the entire training set, the mean of last year recorded on the training set, the last known value of the training set, and we implemented a naïve seasonal forecast. Each baseline was then evaluated on a test set using the MAPE metric. Figure 2.10 summarizes the MAPE of each baseline that we developed throughout this chapter.

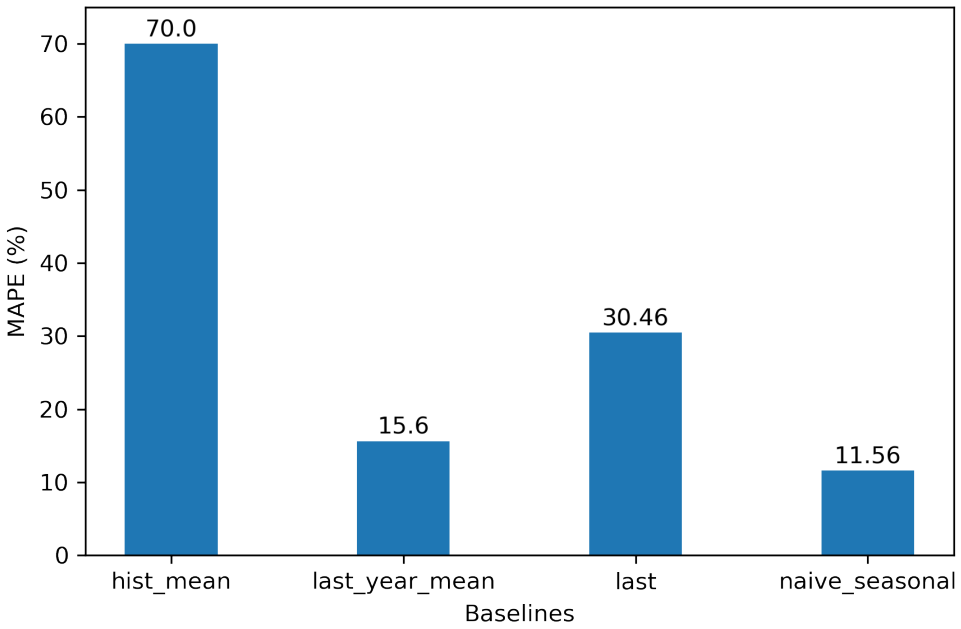


Figure 2.10 MAPE of all four baselines developed in this chapter. The lower the MAPE, the better the baseline; therefore, we choose the baseline using the naïve seasonal forecast as a benchmark and compare it to our more complex models.

As you can see in figure 2.10 the baseline using the naïve seasonal forecast has the lowest MAPE, and therefore the best performance.

Keep in mind that a baseline model serves as a basis for comparison. We then develop more complex models by applying statistical learning or deep learning techniques. After, when we evaluate our more complex solutions against the test set and record our error metric, we can compare it the one from the baseline. In our case, we compare the MAPE from a complex model against the MAPE of our naïve seasonal forecast. If it is lower than 11.56%, then we know that we have a better performing model.

Now, there will be special situations in which a time series can only be forecast using naïve methods. This is a special case where our process moves at random, and therefore cannot be predicted using statistical learning methods. This means that we are in the presence of a random walk, which we will examine in the next chapter.

2.7 Summary

- Time series forecasting starts with a baseline model that serves as a benchmark for comparison with more complex models.
- A baseline model is a trivial solution to our forecasting problem because it only uses heuristics, or simple statistics, such as the mean.
- The MAPE stands for *mean absolute percentage error*, and it is an intuitive measure of how much a predicted value deviates from the actual value.
- There are many ways to develop a baseline: we can use the mean, the last known value, or the last season.

3

Going on a random walk

This chapter covers

- Identifying a random walk process
- Understanding the ACF function
- Understanding differencing, stationarity, and white noise
- Using the ACF plot and differencing to identify a random walk
- Forecasting a random walk

In the previous chapter, we compared different naïve forecasting methods and we learned that they often serve as benchmarks for our more sophisticated models. However, there are instances where the simplest methods will yield the best forecasts. This is the case when we face a random walk process.

In this chapter, we will cover what a random walk process is, how to recognize it, and how to make forecasts using random walk models. Along the way, we will introduce the concepts of differencing, stationarity, and white noise that are used will come back in later chapters as we develop more advanced statistical learning models.

Suppose that you want to buy shares of Alphabet Inc. (GOOGL). Ideally, you would want to buy if the closing price of the stock is expected to go up in the future, otherwise, your investment will not be profitable. Hence, you decide to collect data on the daily closing price of GOOGL over one year and use time series forecasting to determine the future closing price of the stock. We can see the closing price of GOOGL between April 28, 2020, and April 27, 2021 in figure 3.1.

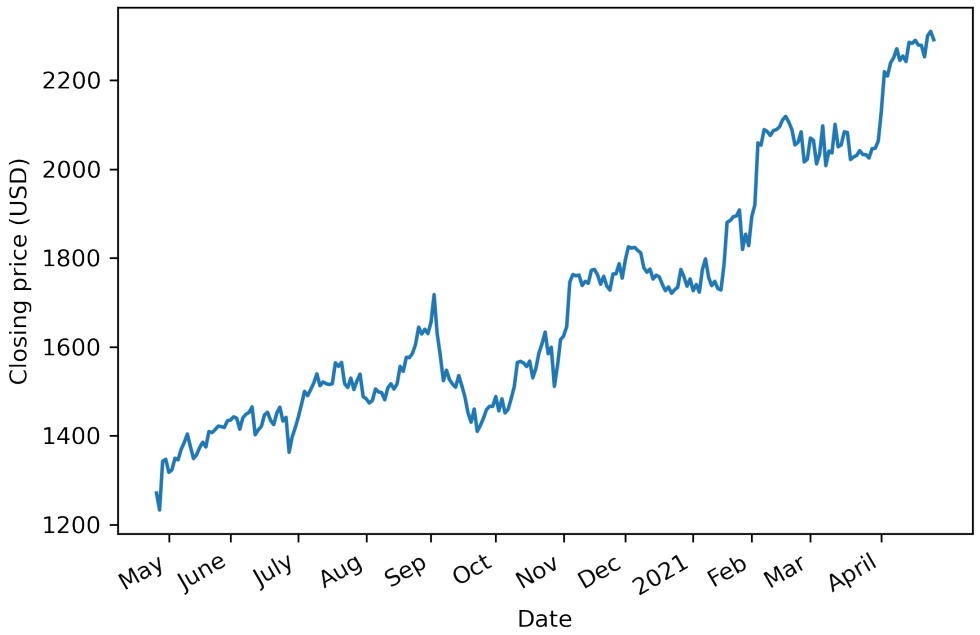


Figure 3.1 Daily closing price of GOOGL between April 28, 2020, and April 27, 2021.

Looking at figure 3.1, we can clearly see a long-term trend, since the closing price increased between April 28, 2020, and April 27, 2021. However, also notice abrupt changes in the trend, with periods where it sharply decreases, before suddenly increasing again.

It turns out that the daily closing price of GOOGL can be modeled using the random walk model. To do so, we will first determine if our process is **stationary** or not. In the event where it is a non-stationary process, we will have to apply transformations, such as **differencing**, in order to make it stationary. Then, we will be able to use the **autocorrelation function** plot to conclude that the daily closing price of GOOGL can be approximated by the random walk model. Both differencing and the autocorrelation plot will be covered in this chapter. Finally, we wrap up the chapter with forecasting methods to predict the future closing price of GOOGL.

By the end of this chapter, you will have mastered the concepts of stationarity, differencing, and autocorrelation, which will come back in later chapters as we further develop our forecasting skills. For now, let's focus first on defining the random walk process.

3.1 The random walk process

A random walk is a process in which there is an equal chance of going up or down by a random number. This is usually observed in financial and economic data, like the daily

closing price of GOOGL. Random walks often expose long periods where a positive or negative trend can be observed. It is often accompanied by sudden changes in direction.

In a random walk process, we say that the present value y_t is a function of the value at the previous timestep y_{t-1}, a constant C, and a random number ϵ_t, also termed as *white noise*. Here, ϵ_t is the realization of a normal distribution with a mean of 0.

Therefore, we can mathematically express a random walk with the following equation, where y_t is the value at the present time *t*, C is a constant, y_{t-1} is the value at the previous timestep *t-1*, and ϵ_t is a random number:

$$y_t = C + y_{t-1} + \epsilon_t$$

Equation 3.1

Note that if the constant C is non-zero, then we designate this process as a random walk with *drift*.

3.1.1 Simulating a random walk process

To help us in our understanding of the random walk process, let's simulate one with Python—that way, we can understand how a random walk behaves and study its properties in a purely theoretical scenario. Then, we will transpose our knowledge onto our real-life example of modeling and forecasting the closing price of GOOGL.

From equation 3.1, we know that a random walk depends on its previous value y_{t-1} plus white noise ϵ_t and some constant C. To simplify our simulation, let's assume that the constant C is 0. That way, our simulated random walk can be expressed as:

$$y_t = y_{t-1} + \epsilon_t$$

Equation 3.2

Now, we must choose the first value of our simulated sequence. Again, for simplification, we initialize our sequence at 0. This will be the value of y_0.

We can now start building our sequence using equation 3.2. We start off with our initial value of 0 at time $t = 0$. Then, from equation 3.2, the value at $t = 1$, represented by y_1 will be equal to the previous value y_0 plus white noise.

$$y_0 = 0$$
$$y_1 = y_0 + \epsilon_1 = 0 + \epsilon_1 = \epsilon_1$$

Equation 3.3

The value at $t = 2$, denoted as y_2 will be equal to the value at the previous step, which is y_1 plus some white noise.

$$y_1 = \epsilon_1$$
$$y_2 = y_1 + \epsilon_2 = \epsilon_1 + \epsilon_2$$

Equation 3.4

Then, the value at $t = 3$, denoted as y_2 will be equal to the value at the previous step, which is y_2 plus some white noise.

$$y_2 = \epsilon_1 + \epsilon_2$$
$$y_3 = y_2 + \epsilon_3 = \epsilon_1 + \epsilon_2 + \epsilon_3$$

Equation 3.5

$$y_t = \sum_{t=1}^{T} \epsilon_t$$

Equation 3.6

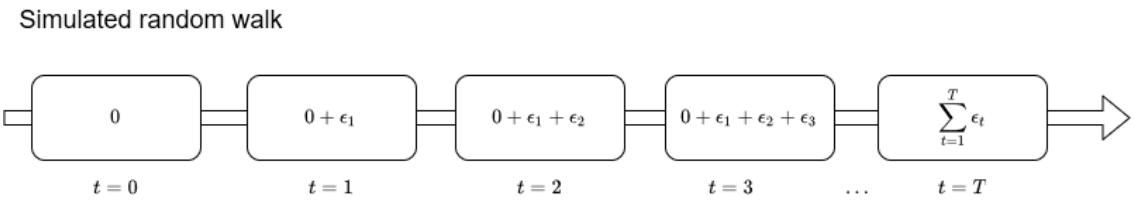


Figure 3.2 Visualizing the construction of our simulated walk. As you can see, our initial value is 0. Then, since the constant was also set to 0, the value of our random walk at any point in time is simply the cumulative sum of random numbers, or white noise.

We are now ready to simulate our random process using Python. In order for this exercise to be reproducible, we will need to set a *seed*. The seed is an integer that we pass to the `random.seed` method. That way, no matter how many times we run the code, the same random numbers will be generated. This ensures that you will obtain the same results and plot as outlined in this chapter.

Then, we must decide on the length of our simulated process. For this exercise, we will generate 1000 samples. The `numpy` library allows us to generate numbers from a normal distribution using the `standard_normal` method. This ensures that the numbers come from a distribution with mean of 0, as per the definition of white noise, and variance of 1. Then, we can set the very first value of our series to 0. Finally, the `cumsum` method will calculate the cumulative sum of white noise for each timestep in our series, and we will have simulated our random walk:

```
import numpy as np

np.random.seed(42)        #A

steps = np.random.standard_normal(1000)     #B
steps[0]=0       #C

random_walk = np.cumsum(steps)     #D
```

#A Set the random seed. This is done by passing an integer, in this case 42. By setting the seed to 42, you can reproduce the results of this chapter, even though the numbers are randomly generated. This ensures that the same random numbers are always obtained. Passing a different integer will generate other values.
#B Generate 1000 random numbers coming from a normal distribution with mean of 0 and variance of 1.
#C Initialize the first value of our series to 0.
#D Calculate the cumulative sum of errors for each timestep in our simulated process

We can plot our simulated random walk and see what it looks like. Since our x-axis and y-axis do not have a real-life meaning, we will simply label them as *timesteps* and *value* respectively. The code block below generates figure 3.3.

```
fig, ax = plt.subplots()

ax.plot(random_walk)
ax.set_xlabel('Timesteps')
ax.set_ylabel('Value')

plt.tight_layout()
```

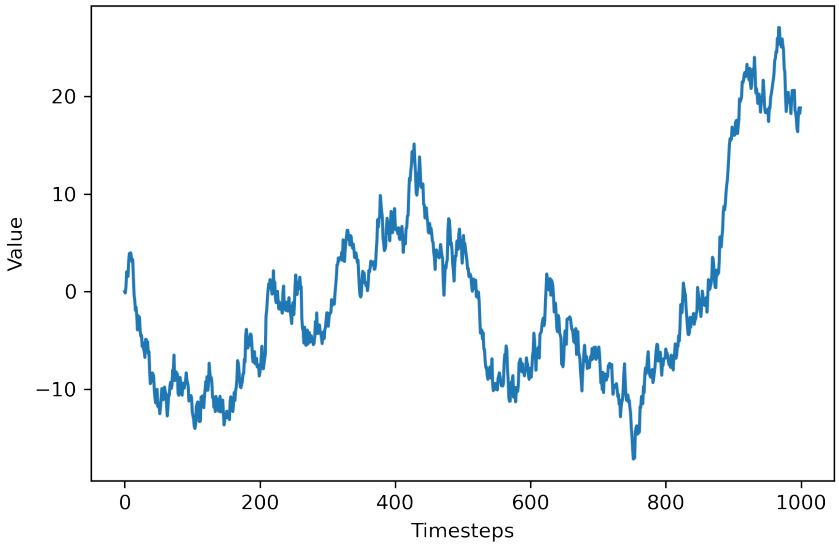


Figure 3.3 Simulated random walk. Notice how we have a positive trend during the first 400 timesteps, followed by a negative trend, and a sharp increase towards the end. These are good hints of a random walk process.

We can see in figure 3.3 the defining characteristics of a random walk. We notice a positive trend over the first 400 timesteps, followed by negative trend, and a sharp increase towards the end. Therefore, we have both sudden changes and long periods where a trend is observed.

Now, we know this is a random walk because we simulated it. However, when dealing with real-life data, we need to find a way to identify whether our time series is a random walk or not. Let's see how can achieve this.

3.2 Identifying a random walk

To determine if our time series can be approximated as a random walk or, we must first define a random walk.

In the context of time series, a random walk is defined as a series whose first difference is stationary and uncorrelated.

Random walk

A random walk is a series whose first difference is stationary and uncorrelated.

This means that the process moves completely at random.

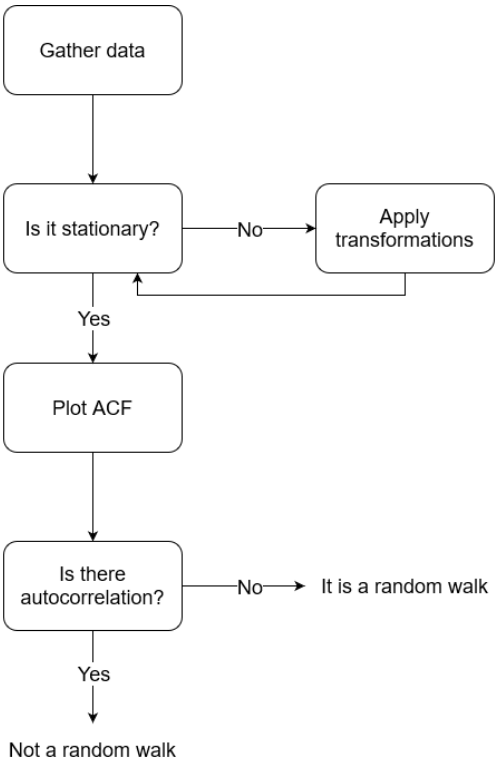


Figure 3.4 Steps to follow to identify if our time series data can be approximated as a random walk or not. The first step is naturally to gather the data. Then, we test for stationarity. If it is not stationary, we apply transformations until stationarity is achieved. Then, we can plot the autocorrelation function (ACF). In the case where there is no autocorrelation, we have a random walk.

The natural first step is to have data, which is the case for us. Then, we test for stationarity. In the case where our data is non-stationary, we will apply a transformation, such as differencing, and test for stationarity again. If it is, we can plot the autocorrelation function (ACF). If the plot does not show significant autocorrelation coefficients, then we can model our time series as a random walk. Otherwise, we are in the presence of another kind of process that we will explore in the next chapter. In the following subsections, we will cover in detail the concepts of stationarity and autocorrelation.

3.2.1 Stationarity

A stationary time series is one whose statistical properties do not change over time. In other words, it has a constant mean, variance, and autocorrelation, and these properties are independent of time.

Many forecasting models assume stationarity. The moving average model (Chapter 4), autoregressive model (Chapter 5), and autoregressive moving average model (Chapter 6) assume stationarity. These models can only be used if we verify that the data is indeed stationary. Otherwise, the models will not be valid, and the forecasts will not be reliable. Intuitively, this makes sense, because if the data is non-stationary, its properties are going change over time, and so it would mean that our model parameters must also change through time. This means that we cannot possibly derive a function of future values as a function of past values, since the coefficients change at each point in time, hence making forecasting unreliable.

Hence, we can see stationarity as an assumption that can make our lives easier when forecasting. Of course, we will rarely see a stationary time series in its original state because we are often interested in forecasting processes with a trend of with seasonal cycles. This is when models like ARIMA (chapter 7) and SARIMA (chapter 8) come into play.

Stationarity

A stationary process is one whose statistical properties do not change over time.

A times series is said to be stationary if its mean, variance and autocorrelation do not change over time.

For now, since we are still in the early stages of time series forecasting, let's focus on stationary time series, which means that we will need to find ways to *transform* our time series to make them stationary. A transformation is simply a mathematical manipulation to the data in order to stabilize its mean and variance, thus making it stationary. The simplest transformation one can apply is differencing. This transformation helps stabilize the mean, which in turn removes or reduces the trend and seasonality effects. Differencing is calculating the series of change from one timestep to another. To accomplish that, we simply subtract the value of the previous timestep y_{t-1} from the value in the present y_t to obtain the differenced value y'_t.

$$y'_t = y_t - y_{t-1}$$

Equation 3.7

Transformation in time series forecasting

A transformation is a mathematical operation applied to a time series in order to make it stationary.

Differencing is a transformation that calculates the change from one timestep to another. This transformation is useful to stabilize the mean.

Applying a log function to the series can stabilize its variance.

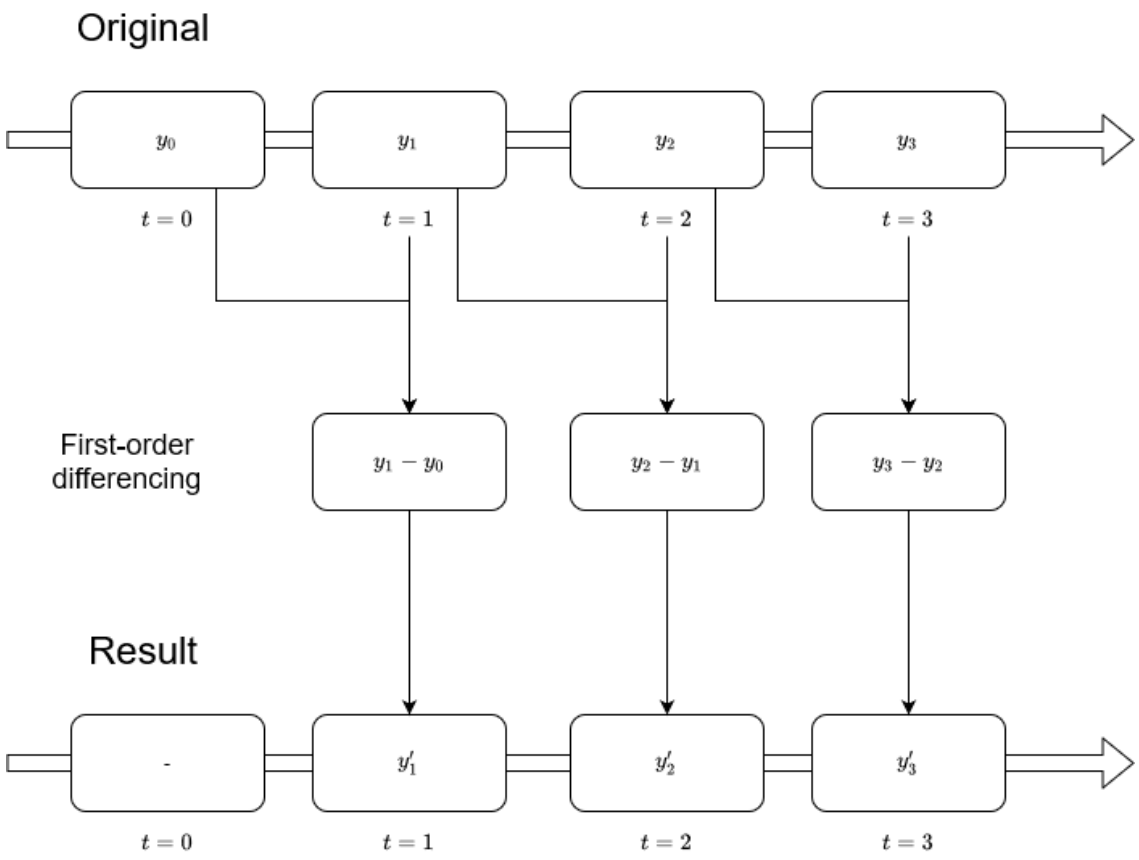


Figure 3.5 Visualizing the differencing transformation. Here, a first-order differencing is applied. Notice how we lose one data point after this transformation because the initial point in time cannot be differenced with previous values since it does not exist.

It is possible to difference a time series many times. Taking the difference once is applying a **first-order** differencing. Taking it a second time, would be a **second-order** differencing. It is often not necessary to difference more than twice to obtain a stationary series.

While differencing is used to obtain a constant mean through time, we must also make sure to have a constant variance in order for our process to be stationary. In such case, logarithms are used to help stabilize the variance.

Keep in mind that when we model a time series that was transformed, we must *untransform* it to bring back the results of the model to the original units of measurement. Therefore, if you apply a log transformation to your data, make sure to raise your forecast values to the power of 10, in order to bring the values back to their original magnitude. That way, our predictions will make sense in their original context.

Now that we know what type of transformations to apply on a time series to make it stationary, we need to find a way to test if a series is stationary or not.

3.2.2 Testing for stationarity

Once a transformation is applied, we need to test for stationarity to determine if we must apply another transformation to make a time series stationary, or if we need to transform it at all. A common test to apply is the augmented Dickey-Fuller (ADF) test.

The ADF test verifies the following null hypothesis: there is a unit root present in a time series. The alternative hypothesis is that there is no unit root, and therefore the time series is stationary. The result of this test is the ADF statistic that is a negative number. The more negative it is, the stronger the rejection of the null hypothesis. In its implementation in Python, the p-value is also returned. If its value is less than 0.05, we can also reject the null hypothesis.

Augmented Dickey-Fuller test (ADF)

The augmented Dickey-Fuller test, or ADF test, helps us determine if a time series is stationary by testing for the presence of a unit root. If a unit root is present, our time series is not stationary.

The null hypothesis states that a unit root is present, meaning that our time series is not stationary.

Let's consider a very simple time series where the present value y_t only depends on its past value y_{t-1} subject to a coefficient α_1, a constant C and white noise ϵ_t, then we can write the following general expression:

$$y_t = C + \alpha_1 y_{t-1} + \epsilon_t$$

Equation 3.8

From equation 3.8, we remember that ϵ_t represents some error that we cannot predict, and C is a constant. Here, α_1 is the root of the time series. This time series will be stationary only

if the root lies within the unit circle. Therefore, its value must be between -1 and 1. Otherwise, the series is non-stationary.

Let's verify this by simulating two different series. One will be stationary and the other will have a unit root, meaning that it will not be stationary. The stationary process follows equation 3.9, and the non-stationary process follows equation 3.10.

$$y_t = 0.5y_{t-1} + \epsilon_t$$

Equation 3.9

$$y_t = y_{t-1} + \epsilon_t$$

Equation 3.10

We can visualize both series in figure 3.6 to gain some intuition about how stationary and non-stationary series evolves through time.

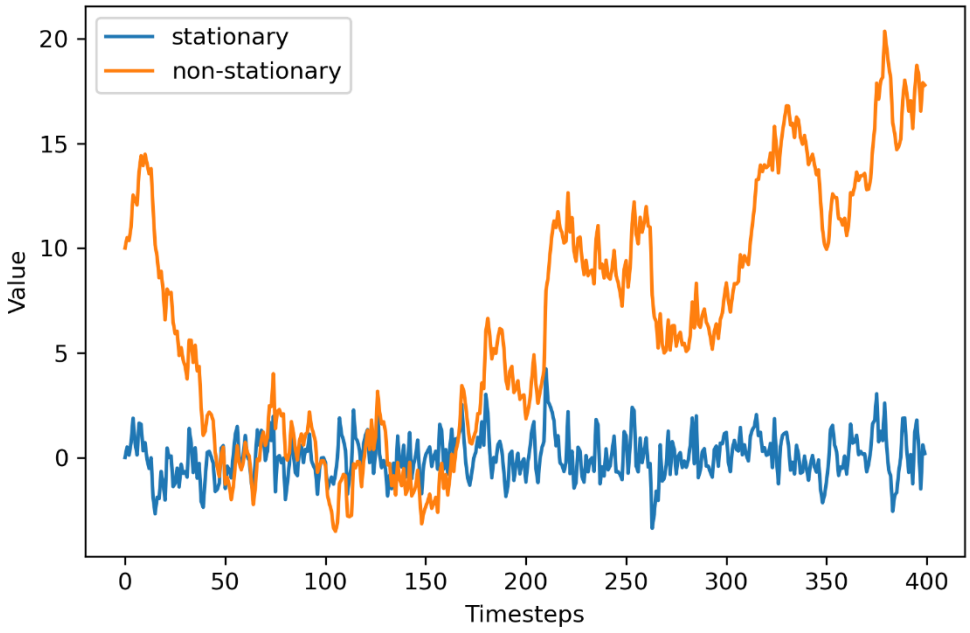


Figure 3.6 Simulated stationary and non-stationary time series over 400 timesteps. We see how the stationary series does not increase or decrease over the long-term. However, the non-stationary process has long periods

From figure 3.6, we can see how the non-stationary process has long period of positive and negative trends. However, the stationary process does not seem to increase of decrease over the long-term. This high-level qualitative analysis can help us intuitively determine if a series is stationary or not.

Now, we remember that a stationary series has constant properties over time, meaning that the mean and variance are not a function of time. Thus, let's plot the mean of each series over time. We expect the mean of the stationary process to be flat over time, whereas the mean of the non-stationary process should vary. We can visualize this in figure 3.7.

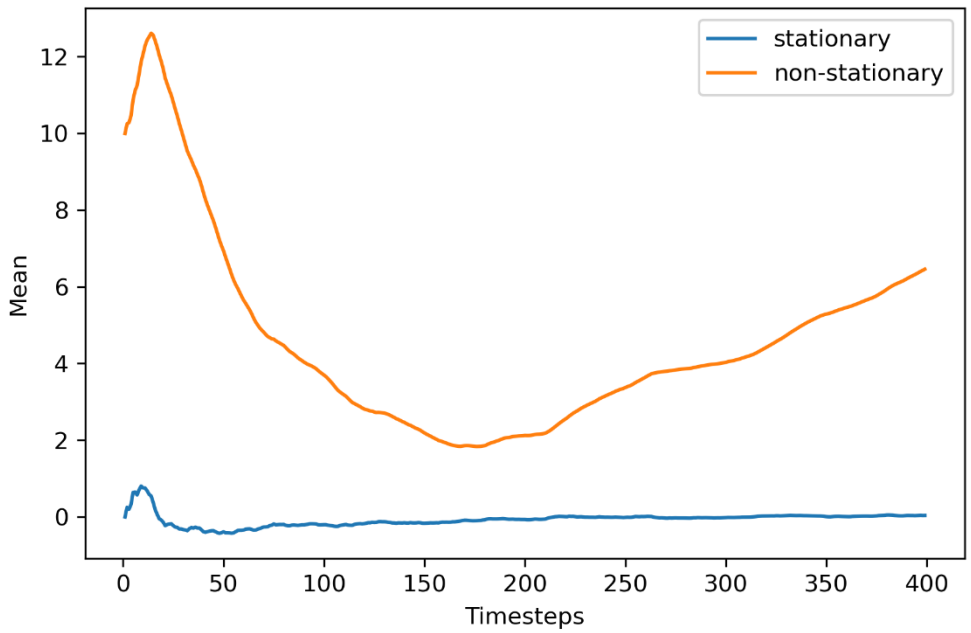


Figure 3.7 Mean of the stationary and non-stationary processes over time. We can see how the mean of the stationary process quickly becomes constant after the first few timesteps. On the other hand, the mean of the non-stationary process is a clear function of time, as it is constantly changing.

Looking at figure 3.7, we can see that the mean of the stationary process quickly becomes constant after the first few timesteps. This is the expected behavior of a stationary process. The fact the mean does not change as a function of time means that it is independent of time, as per the definition of a stationary process. However, the mean of the non-stationary process is clearly a function of time, as we see it decreasing and increasing again over time. Thus, we see how the presence of a unit root makes the mean of the series dependent on time, and so the series is not stationary.

Let's further prove to ourselves that a unit root is a sign of non-stationarity by plotting the variance of each series over time. Again, a stationary series will have a constant variance over time, meaning that it is time independent. On the other hand, the non-stationary process will have a variance that changes over time. We can visualize that in figure 3.8.

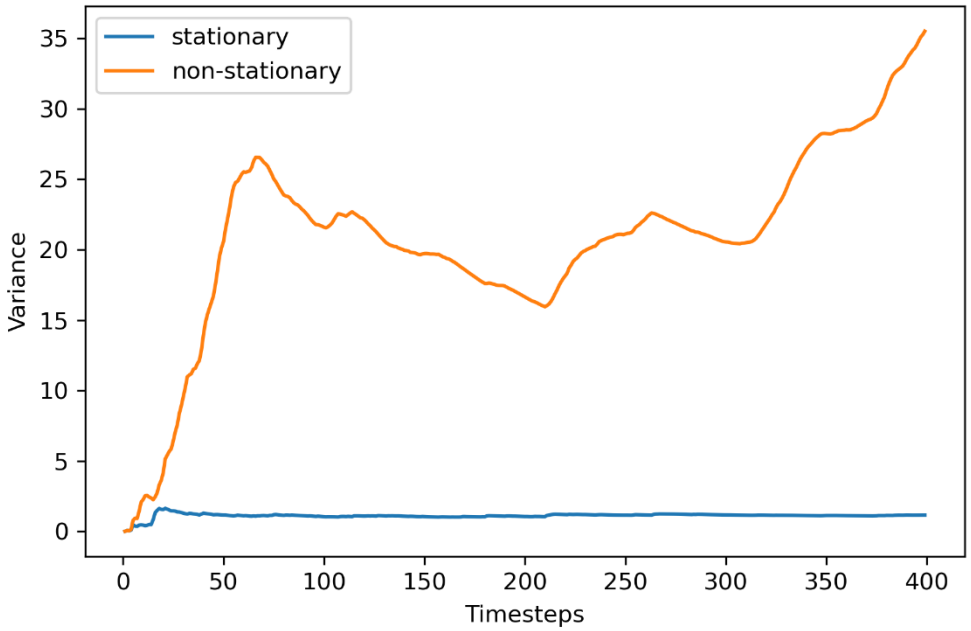


Figure 3.8 Variance of the simulated stationary and non-stationary series over time. Clearly, the variance of the stationary process is independent of time, as we see it is constant after the first few timesteps. For the non-stationary process, the variance indeed changes over time, meaning that it is not independent.

Looking at figure 3.8, we can clearly see that the variance if constant over time, after the first few timesteps, for the stationary process, which follows equation 3.9. Again, this corresponds to the definition of a stationary process since variance does not depend on time. On the other hand, the process with a unit root clearly has a variance that depends on time, since its greatly varies over the 400 timesteps. Therefore, this series is not stationary.

By now, we should be convinced that a series with a unit root is not a stationary series. In both figure 3.7 and 3.8, the mean and variance were respectively dependent on time, as their value kept changing. Meanwhile, the series with a root of 0.5 displayed a constant mean and variance over time, demonstrating that this series is indeed stationary.

All these steps were performed to justify the use of the augmented Dickey-Fuller (ADF) test. We know that the ADF test verifies the presence of a unit root in the series. The null hypothesis stating that a unit root is present, means that the series is not stationary. If the

test returns a p-value less than 0.05, we can then reject the null hypothesis, meaning that there are not unit roots, and so the series is stationary.

Once we have a stationary series, we must then determine if there is autocorrelation or not. Remember that a random walk is a series whose first difference and stationary and uncorrelated. While the ADF test takes care of the stationarity portion, we can use the autocorrelation function to determine if the series is correlated or not.

3.2.3 The autocorrelation function

Once a process is stationary, plotting the autocorrelation function is a great way to understand what type of process we are analyzing. In this case, we will use it to determine if we are studying a random walk or not.

We know that correlation measures the extent of a linear relationship between two variables. Autocorrelation therefore measures the linear relationship between lagged values of a time series. Thus, the autocorrelation function reveals how the correlation between any two values changes as the lag increases. Here, the lag is simply the number of timesteps separating two values.

Autocorrelation function

The autocorrelation function measures the linear relationship between lagged values of a time series.

In other words, it measures the correlation of the time series with itself.

For example, we can calculate the autocorrelation coefficient between y_t and y_{t-1}. In this case, the lag is equal to 1, and the coefficient would be denoted as r_1. Similarly, we can calculate the autocorrelation between y_t and y_{t-2}. Then, the lag would be 2, and the coefficient would be denoted as r_2. When we plot the ACF function, the coefficient is the dependent variable, while the lag is the independent variable. Note that the autocorrelation coefficient at lag 0 will always be equal to 1. This makes sense intuitively, because the linear relationship between a variable and itself at the same timestep should be perfect, and therefore equal to 1.

In the presence of a trend when plotting the ACF, the coefficients will be high for short lags, and we will see a linear decrease as the lag increases. If the data is seasonal, the ACF plot will also display cyclical patterns. Therefore, plotting the ACF function of a non-stationary process will not give us further information as to what we already know by looking at the evolution of our process through time. However, plotting the ACF for a stationary process can for instance help us identify the presence of a random walk.

3.2.4 Putting it all together

Now that we understand what stationarity is, how to transform a time series to make it stationary, what statistical test can be used to assess stationarity, and how plotting the ACF function will help us identify the presence of a random walk, we are ready to put all these

concepts together and apply them in Python. In this section, we will work if our simulated data and cover the necessary steps to identify a random walk.

The first step is to determine if our random walk is stationary or not. Now, we know that since there are visible trends in our sequence, it is not stationary. Nevertheless, let's apply the augmented Dickey-Fuller test to make sure. We will use the `statsmodels` library, which is a Python library that implements many statistical models and tests. To run the ADF test, we simply pass it our array of simulated data. The result is a list of different values, but we are mainly interested in the first two: the ADF statistic and the p-value.

```
from statsmodels.tsa.stattools import adfuller

ADF_result = adfuller(random_walk) #A

print(f'ADF Statistic: {ADF_result[0]}')    #B
print(f'p-value: {ADF_result[1]}') #C
```

#A Pass the simulated random walk to the `adfuller` function
#B Retrieve the ADF statistic, which the first value in the list of results
#C Retrieve the p-value, which is the second value in the list of results

This prints an ADF statistic of -0.97, and a p-value of 0.77. Clearly, the ADF statistic is not a large negative number and with a p-value greater than 0.05, we cannot reject the null hypothesis stating that our time series is not stationary. We can further support our conclusion by plotting the ACF function.

The `statsmodels` library conveniently has a function to quickly plot the ACF. Again, we can simply pass it our array of data. We can optionally specify the number of lags, which will determine the range on the x-axis. In this case, we will plot the first 20 lags, but feel free to plot as many lags as you wish. The output will be figure 3.9.

```
from statsmodels.graphics.tsaplots import plot_acf

plot_acf(random_walk, lags=20);
```

Which results in the figure 3.9:

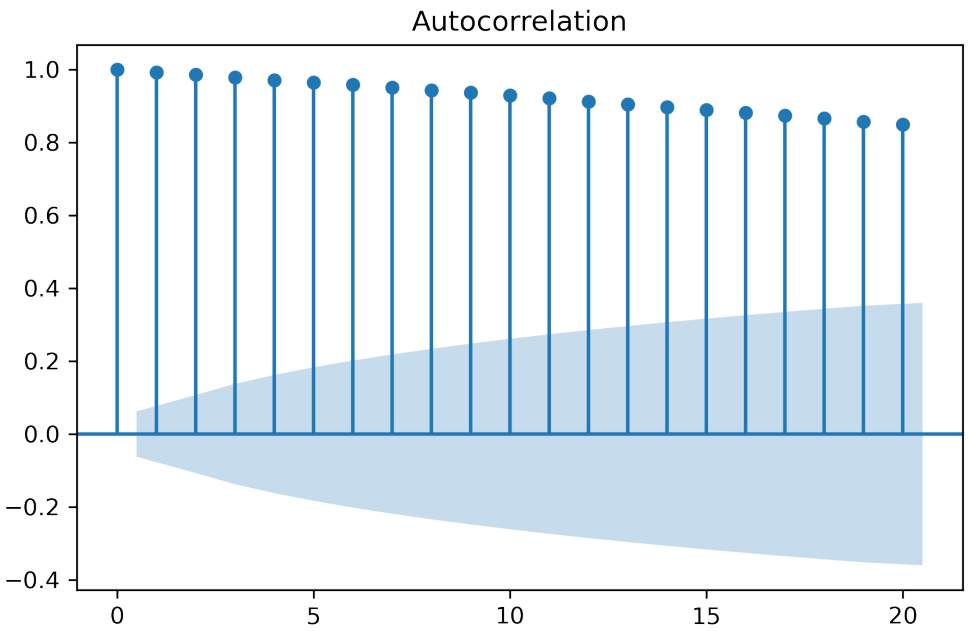


Figure 3.9 Plot of the autocorrelation function of our simulated random walk. Notice how the autocorrelation coefficients slowly decrease. Even at lag 20, the value is still autocorrelated, which means that our random walk is not stationary at the moment.

Looking at figure 3.9, we notice how the autocorrelation coefficients slowly decrease as the lag increases, which is a clear indicator that our random walk is not a stationary process. Note that the shaded area represents a confidence interval. If a point is within the shaded area, then it is not significantly different from 0. Otherwise, the autocorrelation coefficient is significant.

Knowing that our random walk is not stationary, we need to apply a transformation to make it stationary in order to retrieve useful information from the ACF plot. Since our sequence mostly display changes in trend without seasonal patterns, we will apply a first-order differencing. Remember that we lose the first data point every time we difference.

To difference, we will use the `numpy` method `diff`. This will difference a given array of data. The `n` parameter controls how many times the array must be differenced. In this case, we apply a first-order differencing, so the `n` parameter is set to 1:

```
diff_random_walk = np.diff(random_walk, n=1)
```

We can visualize the differenced simulated random walk in figure 3.10.

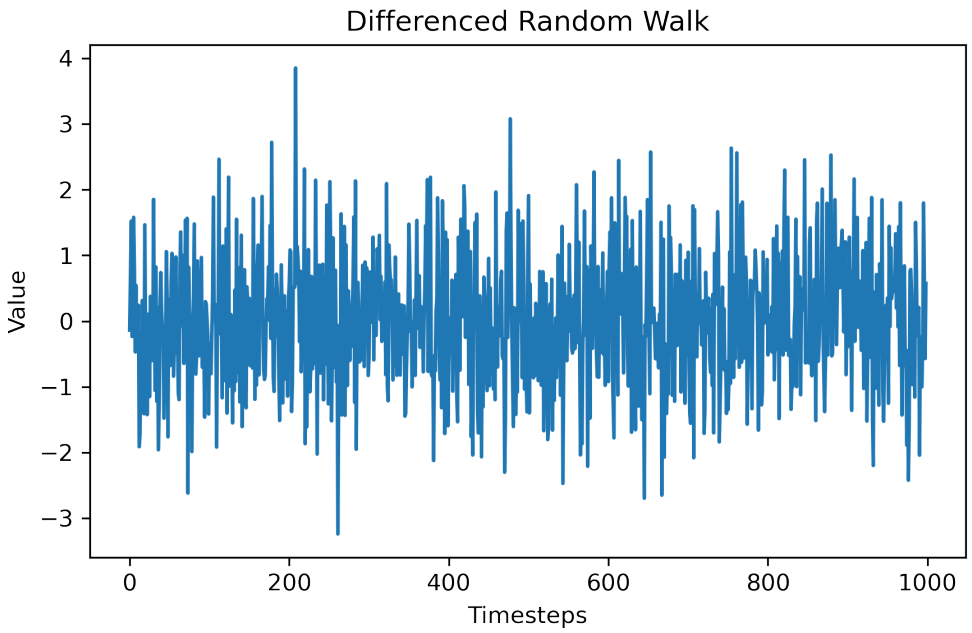


Figure 3.10 Evolution of our differenced random walk. It seems that we successfully removed the trend and that variance is stable.

From figure 3.10, it seems that we successfully removed the trend from our series. Furthermore, the variance looks quite stable. Let's test for stationarity again using the ADF test:

```
ADF_result = adfuller(diff_random_walk)     #A

print(f'ADF Statistic: {ADF_result[0]}')
print(f'p-value: {ADF_result[1]}')
```

#A Here, we pass in our differenced random walk

This prints out an ADF statistic of -31.79 with a p-value of 0. Here, the ADF statistic is a large negative number and the p-value is less than 0.05. Therefore, we reject the null hypothesis, and we can say that this process has no unit root and is thus stationary.

We can now plot the ACF function of our newly stationary series:

```
plot_acf(diff_random_walk, lags=20);
```

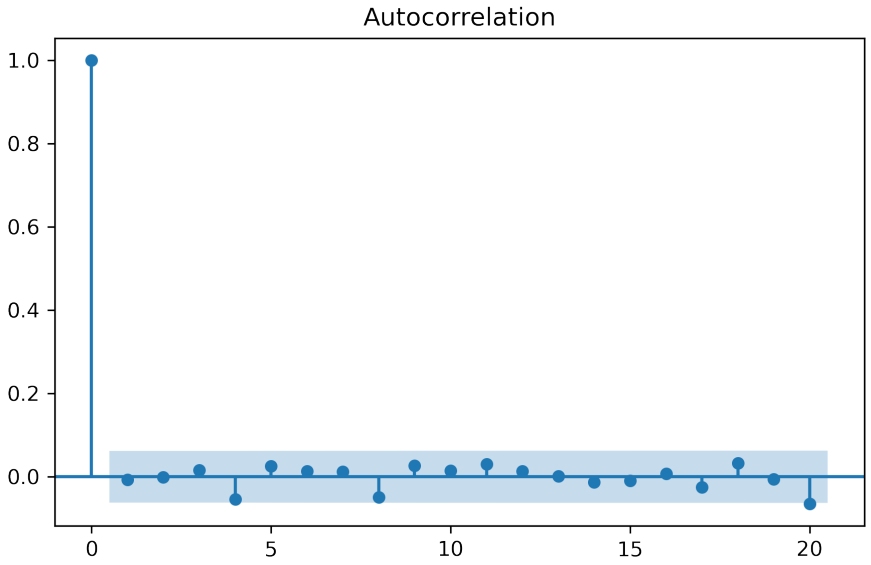


Figure 3.11 ACF plot of our differenced random walk. Notice how there are not significant coefficients after lag 0. This is a clear indicator that we are dealing with a random walk.

Looking at figure 3.11, we notice that there are no significant autocorrelation coefficients after lag 0. This means that the stationary process is completely random and can therefore be described as *white noise*. Each value is simply a random step away from the previous one with no relation between them.

Thus, we have demonstrated that our simulated data is indeed a random walk: he series is stationary and uncorrelated after a first-order differencing, which corresponds to the definition of a random walk.

3.2.5 Is GOOGL a random walk?

Now that we applied the necessary steps to identify a random walk on our simulated data, this is a great time to test our knowledge and new skills on our real-life dataset. Taking the closing price of GOOGL between April 28, 2020, and April 27, 2021, from finance.yahoo.com, let's determine if the process can be approximated as a random walk or not.

You can load the data in a `DataFrame` using the `read_csv` method from `pandas`.

```
df = pd.read_csv('data/GOOGL.csv')
```

Hopefully, your conclusion is that the closing price of GOOGL is indeed a random walk process. Let's see how we arrive at this conclusion. For visualization purposes, let's quickly plot our data, which results in figure 3.12.

```
fig, ax = plt.subplots()
```

```
ax.plot(df.Date, df.Close)
ax.set_xlabel('Date')
ax.set_ylabel('Closing price (USD)')

plt.xticks(
    [4, 24, 46, 68, 89, 110, 132, 152, 174, 193, 212, 235],
    ['May', 'June', 'July', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec', 2021, 'Feb', 'Mar',
       'April']#A

fig.autofmt_xdate()
plt.tight_layout()
```

#A Nicely label the ticks on the x-axis

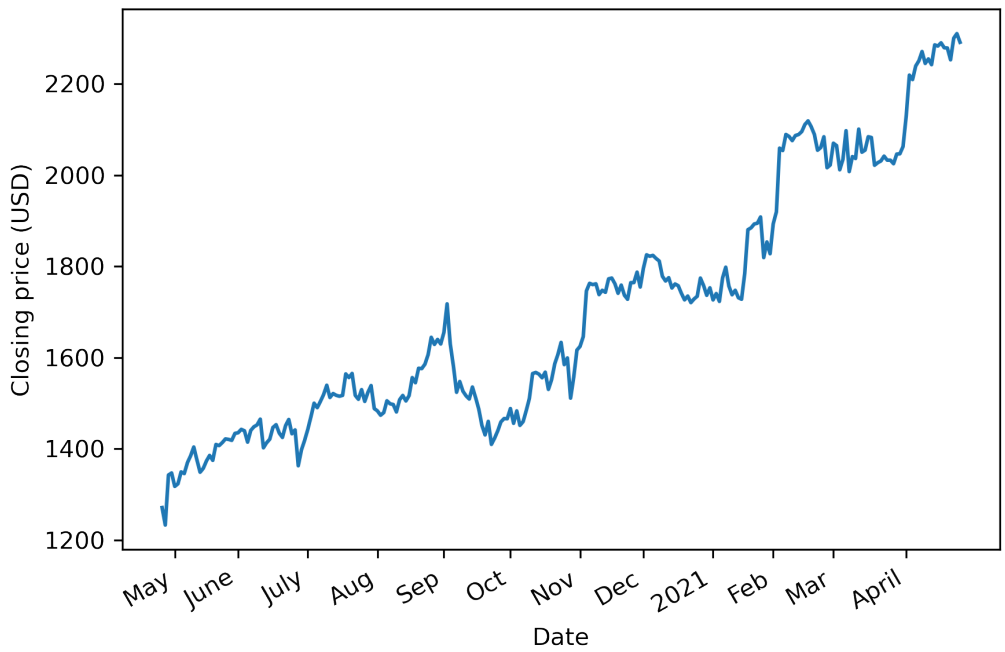


Figure 3.12 Closing price of GOOGL between April 28, 2020, and April 27, 2021.

Looking at figure 3.10, we can clearly see a trend in our data as the closing price is increasing over time; therefore, we do not have a stationary process. This is further supported by the ADF test that returns an ADF statistic of 0.16 and a p-value greater than 0.05 so we know that our data is not stationary.

```
GOOGL_ADF_result = adfuller(df.Close)

print(f'ADF Statistic: {GOOGL_ADF_result[0]}')
```

```
print(f'p-value: {GOOGL_ADF_result[1]}')
```

Hence, we will difference our data in order to see if it makes it stationary.

```
diff_close = np.diff(df['Close'], n=1)
```

Running the ADF test on the differenced data gives an ADF statistic of -5.3 and a p-value smaller than 0.05, meaning that we have a stationary process.

```
GOOGL_diff_ADF_result = adfuller(diff_close)

print(f'ADF Statistic: {GOOGL_diff_ADF_result[0]}')
print(f'p-value: {GOOGL_diff_ADF_result[1]}')
```

Now, we can plot the ACF function in figure 3.13 and see if there is autocorrelation or not.

```
plot_acf(diff_close, lags=20);
```

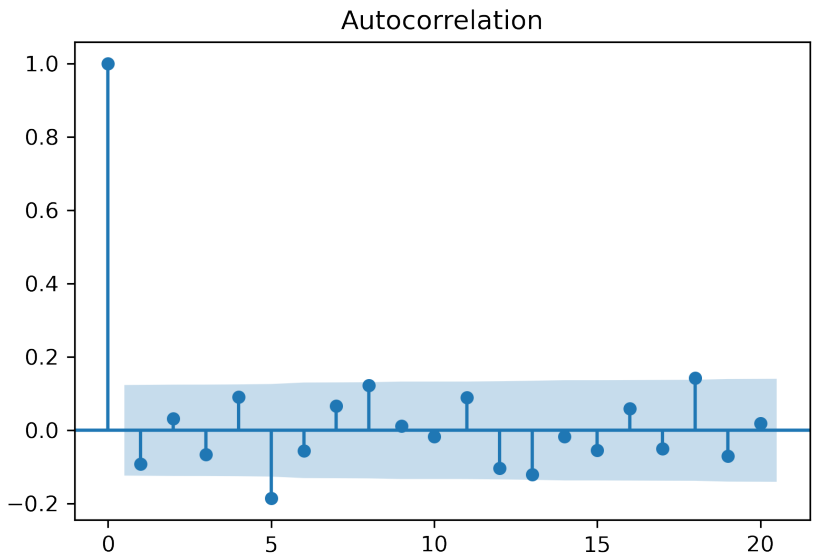


Figure 3.13. We can see that there are no significant coefficients in the ACF plot. You might notice that at lag 5 and 18, the coefficients are significant, while the others are not. This happens by chance with some data and these points can be assumed to be non-significant, because we do not have consecutive significant coefficients between lag 0 and 5, or lag 0 and 18.

Figure 3.13 might make you scratch your head and wonder if there is autocorrelation or not. We do not see any significant coefficients, except at lag 5 and 18. This is a situation that can arise sometimes and is due to chance only. In such situation, we can safely assume that the coefficients at lag 5 and 18 are not significant, because we do not have consecutive

significant coefficients between lag 0 and 5, or between lag 0 and 18. It just happened by chance that the differenced values are slightly correlated with the one at lag 5 and 18.

Therefore, we can conclude that the closing price of GOOGL can be approximated by a random walk process. Taking the first difference makes the series stationary and its ACF plot shows no autocorrelation, meaning that it is purely random.

3.3 Forecasting a random walk

Now that we know what a random walk is and how to identify one, we are ready to start forecasting. This might come as a surprise, since we established that a random walk takes random steps as time progresses. Of course, predicting a random change is impossible, unless we predict a random value ourselves, which is not ideal.

In this case, we can only use naïve forecasting methods, or baselines, that we covered in chapter 2. Since the values change randomly, there is no statistical learning model that can be applied. Instead, we can only reasonably predict the historical mean, or the last value.

Depending on the use case, your forecasting horizon will change. Ideally, when dealing with a random walk, you would only forecast the next timestep. However, it is possible that you are required to forecast many timesteps into the future. Let's see how to tackle each situation.

3.3.1 Forecasting on a long horizon

Let's see how we can forecast a random walk on a long horizon. This is not an ideal case, because a random walk can unexpectedly increase or decrease since past observations are not predictive of the change in the future. We will keep working with our simulated random walk.

To make things easier, we will assign the random walk to a `DataFrame` and split the dataset into a training and test set. The training set will contain the first 800 timesteps, which corresponds to 80% of the simulated data. The test set will thus contain the last 200 values:

```
import pandas as pd

df = pd.DataFrame({'value': random_walk})   #A

train = df[:800] #B
test = df[800:]  #C
```

#A Assign the simulated random walk to a `DataFrame`. It will contain a single column called *value*.
#B The first 80% of the data is assigned to the training set. Since we have 1000 timesteps, 80% of our simulated data corresponds to all the values up to index 800.
#C Assign the last 20% of the simulated random walk to the test set.

Figure 3.14 illustrates our split. Using the training set, we must now predict the next 200 timesteps in the test set.

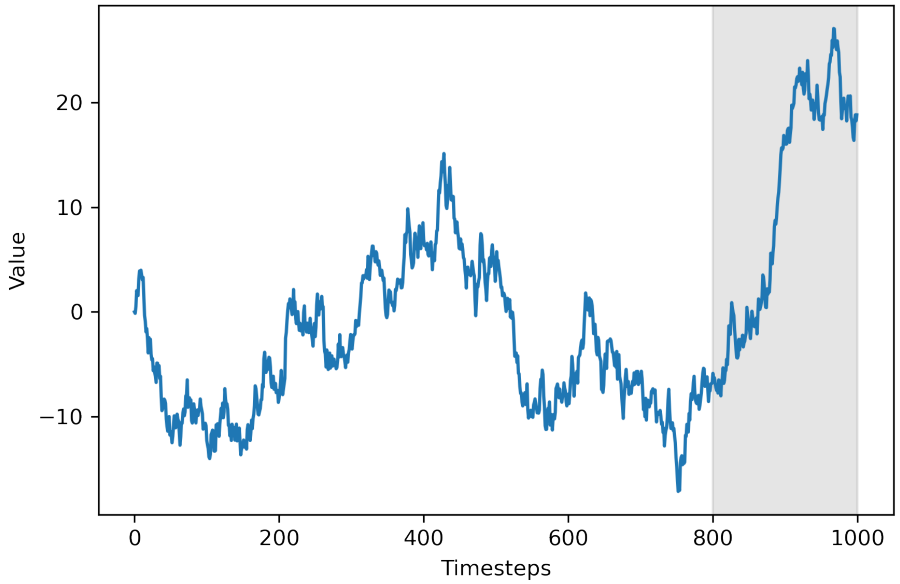


Figure 3.14 Train/test split of our generated random walk. The first 800 timesteps are part of the training set, and the remaining value are part of the test set. Our goal is to forecast the values in the shaded area.

As mentioned, we can only use naïve forecasting methods for this situation, since we are dealing with a random walk. In this case, we will use the historical mean, the last known value, and the drift method.

Forecasting the mean is fairly straightforward. Simply calculate the mean of the training set and we will say that the next 200 timesteps will be equal to that value. Here, we will create a single `DataFrame` that will hold the real observed values as well as our predictions from the baselines. To do so, we will assign a copy of the test set to a `DataFrame` called `pred_df`. That way, we have the observed values. Then, we can assign our historical mean prediction to the *pred_mean* column.

```
mean = np.mean(train.value)        #A

pred_df = test.copy()     #B
pred_df['pred_mean'] = mean        #C

pred_df.head()    #D
```

#A Calculate the mean of the training set
#B Create a DataFrame to hold the observed values and our predictions. At this point, only the observed values are in the DataFrame.
#C Predict the historical mean for the next 200 timesteps
#D Show the first five rows of `pred_df`

You will get an historical mean of -3.68. This means that we forecast that the value for the next 200 timesteps of our simulated random walk will be equal to -3.68.

Another possible baseline is to predict the last known value of the training set. Here, we simply need to extract the last value of the training set and assign its value as our prediction for the next 200 timesteps.

```
last_value = train.iloc[-1].value  #A
pred_df['pred_last_value'] = last_value     #B

pred_df.head()
```

#B Assign the last value as a prediction for the next 200 timesteps under the column *pred_last_value*

Finally, we apply the drift method, which we have not covered yet. The drift method is a modification to predicting the last known value. Here, we allow the values to increase or decrease over time. The rate at which values will change in the future is equal to that seen in the training set. Therefore, it is equivalent to calculating the slope between the first and last value of the training set, and simply extrapolating this straight line into the future.

Remember that we can calculate the slope of a straight line by dividing the change in the y-axis by the change in the x-axis. In our case, the change in the y-axis is the difference between the last value of our random walk y_f and its initial value y_i. Then, the change in the x-axis is equivalent to the number of timesteps, as shown in equation 3.11.

$$slope = \frac{\Delta y}{\triangle x} = \frac{y_f - y_i}{\#timesteps}$$

Equation 3.11

We calculated the last value of the training set when we implemented the last-value baseline and we know that the initial value of our simulated random walk is 0; therefore, we can plug the numbers in equation 3.9 and calculate the drift in equation 3.12.

$$drift = \frac{-6.81 - 0}{800} = -0.0085$$

Equation 3.12

Let's implement this in Python now. We will also calculate the change in the x-axis and the y-axis, and simply divide them to obtain the drift:

```
deltaX = 800 - 0  #A
deltaY = last_value - 0   #B

drift = deltaY / deltaX   #C
```

```
print(drift)
```

#A Calculate the change in the x-axis, which is equivalent to the number of timesteps in the training set
#B Calculate the difference between the last and initial value of the simulated random walk in the training set. Recall that the last value of the training set is in the `last_value` variable from the previous baseline we implemented.
#C Calculate the drift according to Equation 3.9.

As expected, this gives us a drift of -0.0085, which means that the values of our forecasts will slowly decrease over time. The drift method simply states that the value of our forecast is linearly dependent on the timestep, the value of the drift, and the initial value of our random walk as expressed in equation 3.13. Keep in mind that our random walk starts at 0, so we can remove that from equation 3.13.

$$\text{forecast} = \text{drift} \times \text{timestep} + y_i$$
$$\text{forecast} = \text{drift} \times \text{timestep}$$

Equation 3.13

Since we want to forecast the next 200 timesteps following the training set, we first create an array containing the range of timesteps starting at 800 and ending at 1000 with a step of 1. Then we simply multiply each timestep by the drift to get our forecast values. Finally, we assign them to the *pred_drift* column of `pred_df`.

```
x_vals = np.arange(800, 1001, 1)   #A

pred_drift = drift * x_vals        #B

pred_df['pred_drift'] = pred_drift #C

pred_df.head()
```

#A Create a list containing the range of timesteps starting at 800 and ending at 1000 with a step of 1.
#B Multiply each timestep by the drift to get the forecast value at each timestep.
#C Assign our forecast values to the *pred_drift* column.

With all three methods used, we can now visualize what our forecasts look like against the actual values of the test set.

```
fig, ax = plt.subplots()

ax.plot(train.value, 'b-')#A
ax.plot(pred_df.value, 'b-')       #B
ax.plot(pred_df.pred_mean, 'r-.', label='Mean')     #C
ax.plot(pred_df.pred_last_value, 'g--', label='Last value')  #D
ax.plot(pred_df.pred_drift, 'k:', label='Drift')    #E

ax.axvspan(800, 1000, color='#808080', alpha=0.2)   #F
ax.legend(loc=2) #G

ax.set_xlabel('Timesteps')
ax.set_ylabel('Value')
```

```
plt.tight_layout()
```

#A Plot the values in the training set

#C Plot the forecast from the historical mean. It will be a red dotted and dashed line.
#D Plot the forecast from the last value of training set. It will be a green dashed line.
#E Plot the forecast using the drift method. It will be a black dotted line.
#F Shade the forecast horizon
#G Place the legend in the upper left corner

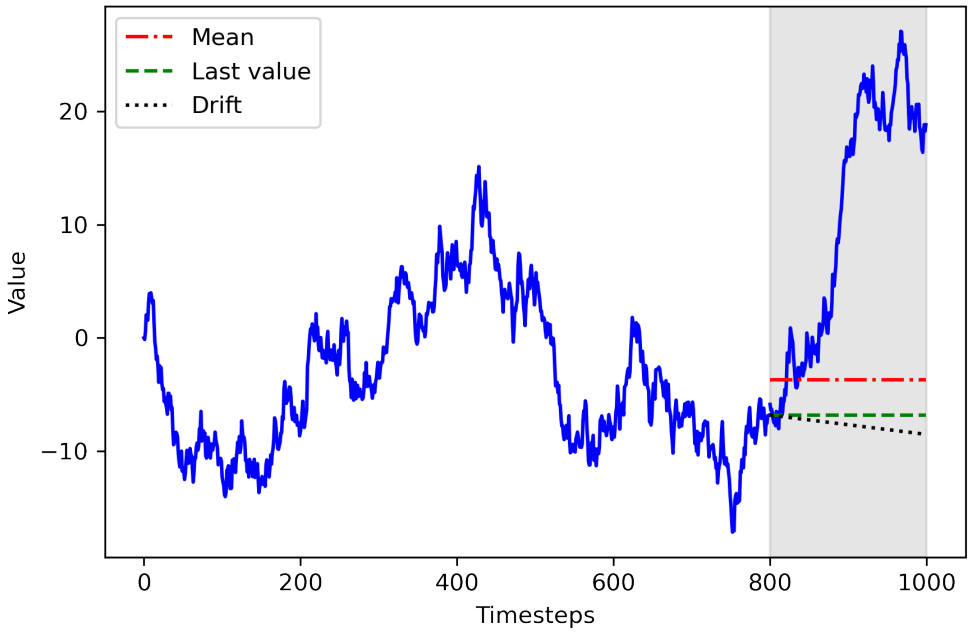


Figure 3.15 Forecasting our random walk using the mean, last known value, and drift method. As you can see, all predictions are fairly poor and fail to predict the sudden increase observed in the test set.

As we can see in figure 3.15, our forecasts are faulty and they all fail to predict the sudden increase observed in the test set, which makes sense because the future change in a random walk is completely random, and therefore unpredictable.

We can further demonstrate that by calculating the mean squared error (MSE) of our forecasts. Note that we cannot use the MAPE as in chapter 2, because our random walk can take the value 0. Of course, it is impossible to calculate the percentage difference from an observed value of 0, because it implies a division by 0, which is not allowed in mathematics.

Therefore, we opt for the MSE, as it can measure the quality of the fit of a model, even if the observed value is 0. The `sklearn` library has a `mean_squared_error` function that simply need the observed values and the predicted values. It will then return the MSE.

```
from sklearn.metrics import mean_squared_error
```

```
mse_last = mean_squared_error(pred_df.value, pred_df.pred_last_value)
mse_drift = mean_squared_error(pred_df.value, pred_df.pred_drift)

print(mse_mean, mse_last, mse_drift)
```

You will obtain an MSE of 327, 425 and 465 for the historical mean, last value, and drift method respectively. We can compare the MSE for each baseline in figure 3.16.

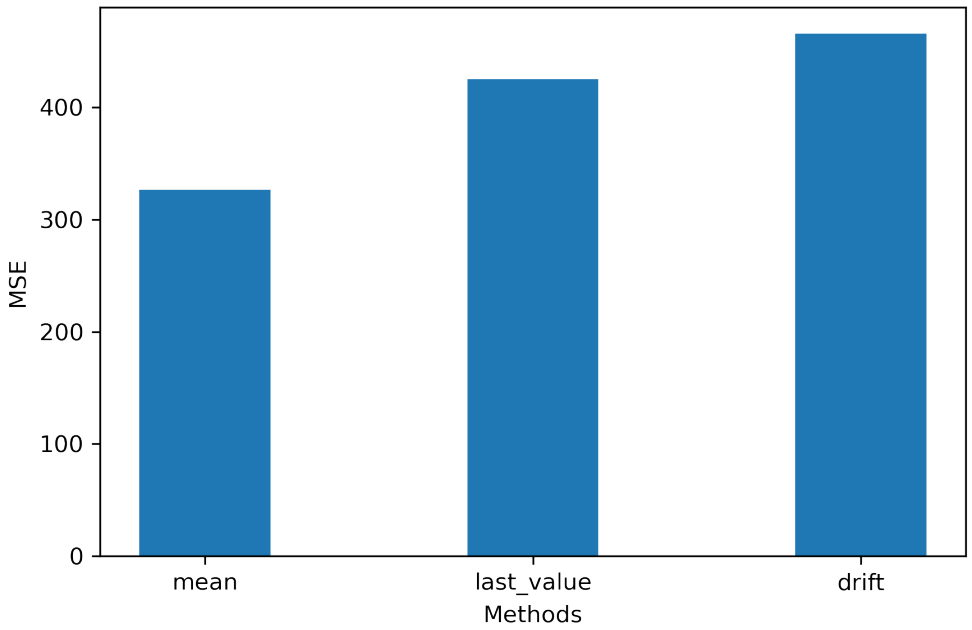


Figure 3.16 MSE of our forecasts. Clearly, the future of a random walk is unpredictable, with MSEs exceeding 300.

From figure 3.16, the best forecast was obtained by predicting the historical mean and yet the MSE exceeds 300. This is an extremely high value considering that our simulated random walk does not exceed the value of 30.

By now, you should be convinced that forecasting a random walk on a long horizon does not make sense. Since the future value is dependent on the past value plus a random number, the randomness portion is magnified in a long horizon since many random numbers are added over the course of many timesteps.

3.3.2 Forecasting the next timestep

Forecasting the next timestep of a random walk is the only reasonable situation we can tackle, although we will still use naïve forecasting methods. Specifically, we will predict the last known value. However, we will make this forecast only for next timestep. That way, our forecast should only be off by a random number, since the future value of a random walk is always the past value plus white noise.

Implementing this method is straightforward: we take our initial observed value and use it to predict the next timestep. Once we record a new value, it will be used as a forecast for the following timestep. This process is then repeated into the future.

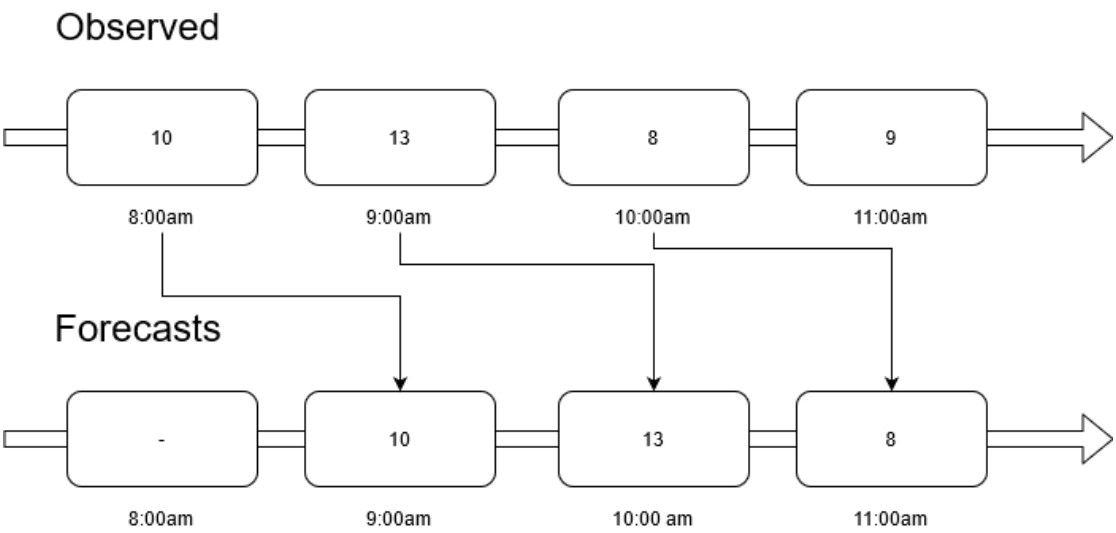


Figure 3.17 Forecasting the following timestep of a random walk. Here, the observed value at point in time will be used as a forecast for the next point in time.

Figure 3.17 illustrates this process. Here, the observed value at 8:00am is used to forecast the value for 9:00am, the actual value observed at 9:00am is used to forecast the value at 10:00am and so on.

Let's apply this method on our random walk process. For the sake of illustrating this method, we will apply it over the entire random walk. This shows how this naïve forecast can look deceptively amazing, when we are actually only predicting the last known value at each timestep.

A good way to simulate this process is by shifting our data. The `pandas` library has a `shift` method that does exactly what we want. We simply pass in the number of period, which in our case is 1, since we are forecasting the next timestep.

```
df_shift = df.shift(periods=1)     #A
```

```
df_shift.head()
```

#A `df_shift` **is now our forecast over the entire random walk, and it corresponds to the last known value at each timestep.**

You will notice that at step 1, the value is 0, which corresponds to the observed value at step 0 in the simulated random walk. Therefore, we are effectively using the present observed value as a forecast for the next timestep. Plotting our forecast yields figure 3.18.

```
fig, ax = plt.subplots()

ax.plot(df, 'b-', label='actual')
ax.plot(df_shift, 'r-.', label='forecast')

ax.set_xlabel('Timesteps')
ax.set_ylabel('Value')

plt.tight_layout()
```

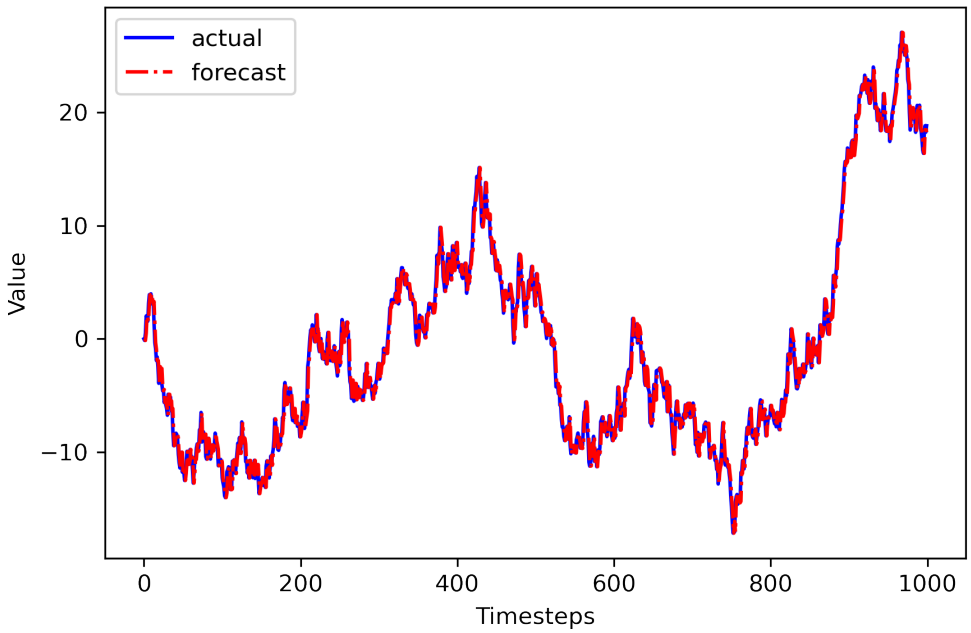


Figure 3.18 Naïve forecast of the next timestep of a random walk. This plot gives the illusion of a very good model, when we are in fact only predicting the value observed at the previous timestep.

Looking at figure 3.18, we might think that we developed an amazing model that is almost a perfect fit to our data. It seems that we do not have two separate lines in the graph, since

```
mse_one_step = mean_squared_error(pred_df.value, df_shift[800:])      #A

print(mse_one_step)
```

#A Calculate the MSE on the test set.

This yields a value of 0.93, which again might lead us to think that we have a very performant model since the MSE if very close to 0. However, we know that we are simply forecasting the value observed at the previous timestep. This becomes more apparent if we zoom in on our graph as shown in figure 3.19.

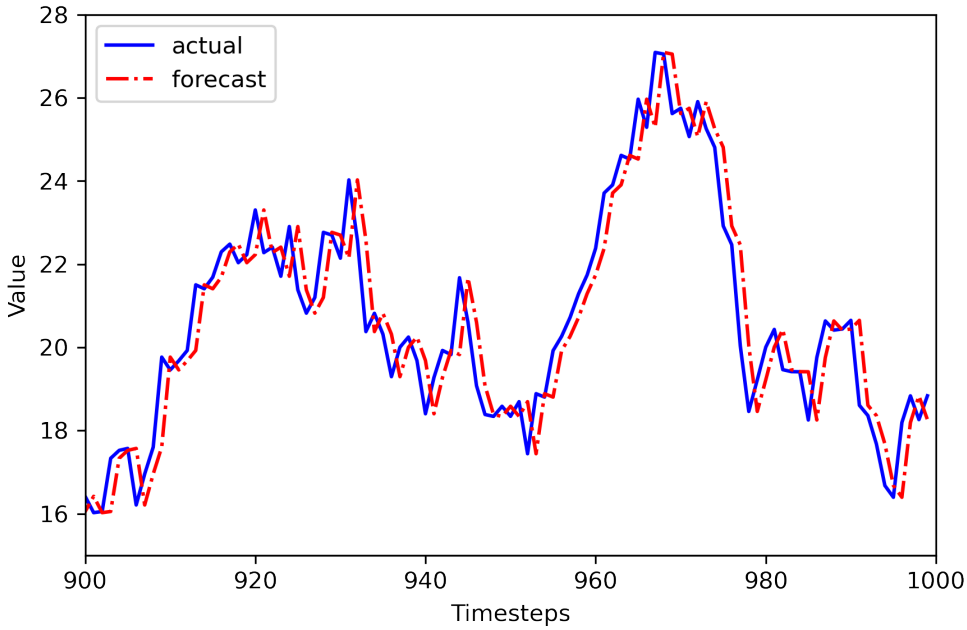


Figure 3.19 Close-up on the last 100 timesteps of our random walk. Here, we can clearly see how our forecasts are a simple shift of the original time series.

Therefore, if a random walk process must be forecast, it is better to make many short-term forecasts. That way, we do not allow for many random numbers to accumulate over time, which will degrade the quality of our forecasts in the long-term. On the other hand, forecasting the next time step prevents magnifying the effect of randomness in the long-term, hence resulting in better forecasts.

Still, because a random process takes random steps into the future, we cannot use statistical or deep learning techniques to fit such a process because there is nothing to learn from randomness and we cannot predict randomness. Instead, we rely on naïve forecasting methods.

3.4 Exercises

Now is a great time to apply the different skills you learned in this chapter. Below are three exercises that will test your knowledge and understanding of random walk and forecasting a random walk. The exercises are in order of difficulty and time required to complete. The solution to exercise 3.4.2 is provided on Github.

3.4.1 Easy: simulate and forecast a random walk

Simulate a different random walk from what we have worked with in this chapter. You can simply change the seed and get new values.

- Generate a random walk of 500 timesteps. Feel free to choose an initial value different from 0. Also, make sure to change the seed by passing a different integer to `np.random.seed()`.
- Plot your simulated random walk
- Test for stationarity
- Apply a first-order difference
- Test for stationarity
- Split your simulated random walk into a training set containing the first 400 timesteps. The remaining 100 timesteps will be your test set.
- Apply different naïve forecasting methods and measure the MSE. Which method yields the lowest MSE?
- Plot your forecasts
- Forecast the next timestep over the test set and measure the MSE. Did it decrease?
- Plot your forecasts

3.4.2 Medium: Forecast the daily closing price of GOOGL

Using the GOOGL dataset that we have worked with in this chapter, apply the forecasting techniques and measure their performance.

- Keep the last 5 days of data as a test set. The rest will be the training set.
- Forecast the last 5 days using naïve forecasting methods and measure the MSE. Which method is the best?
- Plot your forecasts
- Forecast the next timestep over the test set and measure the MSE. Did it decrease?
- Plot your forecasts

3.4.3 Hard: Forecast the daily closing price of a stock of your choice

The historical daily closing price of many stocks is available for free on finance.yahoo.com. Select a stock ticker of your choice and download its historical daily closing price of 1 year.

- Plot the daily closing price of your chosen stock
- Determine if it is a random walk or not.
- If it is not a random walk, explain why.
- Keep the last 5 days of data as a test set. The rest will be the training set.
- Forecast the last 5 days using naïve forecasting methods and measure the MSE. Which method is the best?
- Plot your forecasts
- Forecast the next timestep over the test set and measure the MSE. Did it decrease?
- Plot your forecasts

3.5 Next steps

In the last chapters, we learned how to develop baseline models and we discovered that in the presence of a random walk, we can only reasonably apply baseline models to make forecasts, since we cannot fit a statistical model or use deep learning techniques on data that takes random steps in the future. Ultimately, we cannot predict random movements.

We learned that a random walk is a sequence where the first difference is a stationary process, meaning that its mean, variance, and autocorrelation are constant over time and is not autocorrelated. The steps required to identify a random walk are shown in figure 3.20.

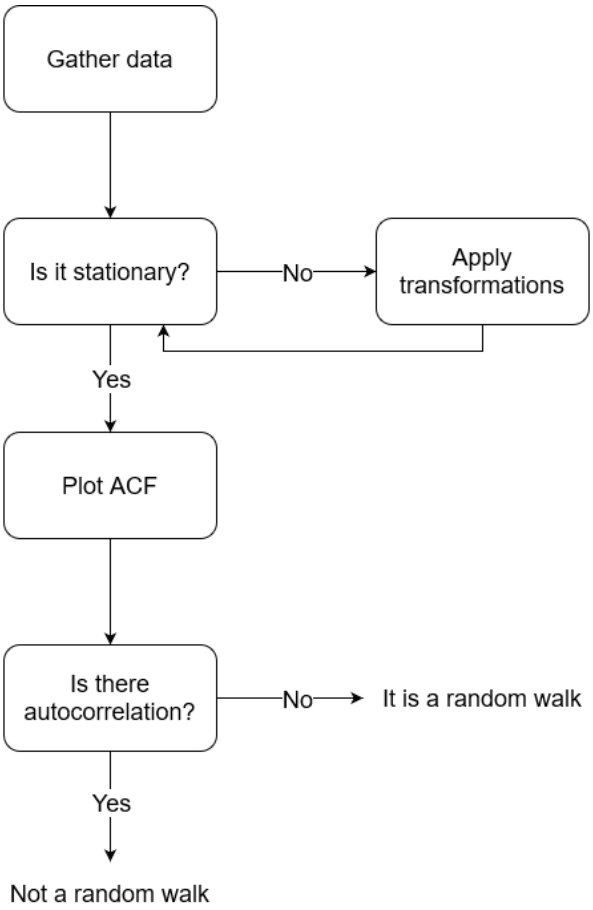


Figure 3.20 Steps to identify a random walk

But what happens if our process is stationary and autocorrelated meaning that we see consecutive significant coefficients on the ACF plot? For now, figure 3.18 simply states that it is not a random walk so we have to find another model to approximate our process and forecast it. In such a situation we are facing a process that can be approximated by the moving average model (MA), an autoregressive model (AR), or the combination of both processes leading to an autoregressive moving average model (ARMA).

In the next chapter, we will focus solely on the moving average model to learn how to identify such process and how to use the moving average model to make forecasts.

3.6 Summary

- A random walk is a process where the first difference is stationary and not

autocorrelated.

- We cannot use statistical or deep learning techniques on a random walk since it moves at random in the future. Therefore, we must use naïve forecasts.
- A stationary time series is one whose statistical properties (mean, variance, autocorrelation) do not change over time.
- The augmented Dickey-Fuller (ADF) test is used to assess stationarity by testing for unit roots.
- The null hypothesis of the ADF test is: there is a unit root in the series. If the ADF statistic is a large negative value and the p-value is less than 0.05, the null hypothesis is rejected, and the series is stationary.
- Transformations are used to make a series stationary. Differencing can stabilize the trend and seasonality, while logarithms stabilize the variance.
- Autocorrelation measures the correlation between a variable and itself at a previous timestep (lag). The autocorrelation function (ACF) shows how the autocorrelation changes as a function of the lag.
- Ideally, we forecast a random walk in the short-term or next timestep. That way, we do not allow for random number to accumulate, which will degrade the quality of our forecasts in the long-term.

4

Modeling a moving average process

This chapter covers

- Defining a moving average process
- Using the ACF to identify the order of a moving average process
- Forecasting a time series using the moving average model

In the previous chapter, we learned how to identify and forecast a random walk process. We defined a random walk process as a series whose first difference is stationary with no autocorrelation. This means that plotting its ACF will show no significant coefficients after lag 0. However, it is possible that a stationary process still exhibits autocorrelation. In this case, we have a time series that can be approximated by a moving average model MA(q), an autoregressive model AR(p), or an autoregressive moving average model ARMA(p,q). In this chapter, we will first focus on identifying and modeling using the moving average model.

Suppose that you want to forecast the volume of sales of widgets from the XYZ Widget Company. By predicting futures sales, the company can better manage its production of widgets to avoid producing too much or too little. In the case where not enough widgets are produced, the company would not be able to meet the client's demand, leaving customers unhappy. On the other hand, producing too many widgets will increase inventory. The widgets might become obsolete or lose their value, which will increase the business's liabilities ultimately making shareholders unhappy.

In this situation, we will specifically study the sales of widgets over 500 days starting in 2019. We can see the recorded sales over time in Figure 4.1. Note that the volume of sales is expressed in thousands of US dollars.

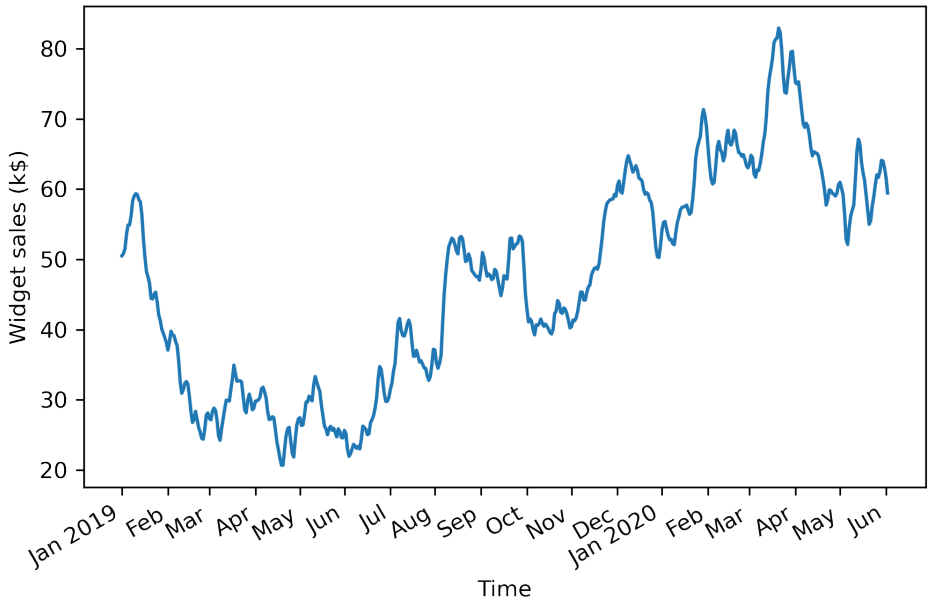


Figure 4.1 Volume of widget sales for the XYZ Widget Company over 500 days, starting in January 1, 2019. Note that this is fictional data but it will be useful for learning how to identify and model a moving average process.

Looking at figure 4.1, we can see a long-term trend with peaks and troughs along the way. From this observation, we can intuitively say that this time series is not a stationary process since we observe a trend over time. Furthermore, there is no apparent cyclical pattern in the data so we can rule out any seasonal effects for now.

In order to forecast the volume of widget sales, we need to identify the underlying process. To do so, we will apply the same steps that we covered in chapter 3 when working with a random walk process, as shown in figure 4.2.

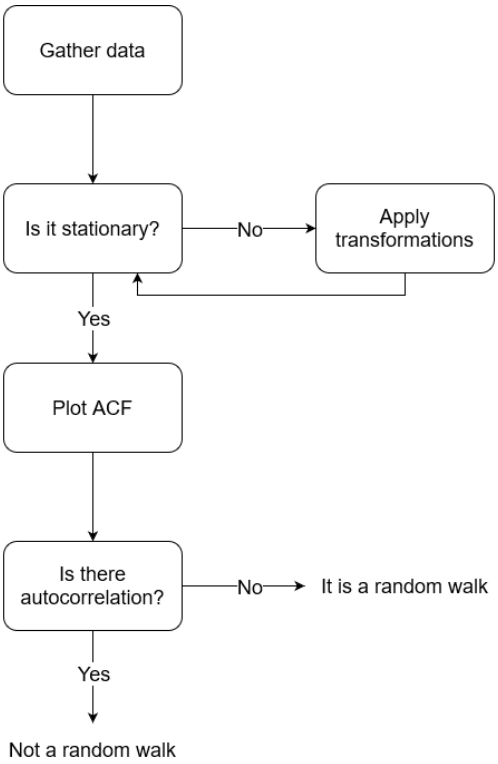


Figure 4.2 Steps to identify a random walk

Once the data is gathered, we will test for stationarity. In the case where it is not stationary, we will apply a transformation to make it stationary. Then, once the series is a stationary process, we will plot the autocorrelation function, or ACF. In our situation of forecasting sales widget, it turns out that our process will show significant coefficients in the ACF plot, meaning that it cannot be approximated by the random walk model.

As we will cover in this chapter, we will find out that the volume of sales of widgets from the XYZ Widget Company can be approximated as a moving average process and we will learn the definition of the moving average model. Then, we show how to identify the order of the moving average process using the ACF plot. The order of the moving average process determines the number of parameters for our model. Finally, we will apply the moving average model to forecast the next 50 days of widget sales.

4.1 Defining a moving average process

A moving average process, or the moving average model, states that the current value is linearly dependent on the current and past error terms. Again, the error terms are assumed to be mutually independent and normally distributed, just like white noise.

A moving average model is denoted as MA(q) where q is the order. The model expresses the present value as a linear combination of the mean of the series μ, the present error term ϵ_t, and past error terms ϵ_{t-q}. The magnitude of the impact of past errors on the present value is quantified using a coefficient denoted as θ_q. Mathematically, we express a general moving average process of order q as in equation 4.1.

$$y_t = \mu + \epsilon_t + \theta_1\epsilon_{t-1} + \theta_2\epsilon_{t-2} + \cdots + \theta_q\epsilon_{t-q}$$

Equation 4.1

Moving average process

A moving average process states that the current value depends linearly on the mean of the series, the current error term, and past error terms.

The moving average model is denoted as MA(q), where q is the order. The general expression of a MA(q) model is:

$$y_t = \mu + \epsilon_t + \theta_1\epsilon_{t-1} + \theta_2\epsilon_{t-2} + \cdots + \theta_q\epsilon_{t-q}$$

The order q of the moving average model determines the number of past error terms that affect the present value. For example, if it is of order one, meaning that we have a MA(1) process, then the model is expressed as equation 4.2. Here, we see how the present value y_t is dependent on the mean μ, the present error term ϵ_t and the error term at the previous timestep $\theta_1\epsilon_{t-1}$.

$$y_t = \mu + \epsilon_t + \theta_1\epsilon_{t-1}$$

Equation 4.2

If we have a moving average process of order two, or MA(2), then y_t is dependent on the mean of the series μ, the present error term ϵ_t, the error term at the previous timestep $\theta_1\epsilon_{t-1}$, and the error term two timesteps prior $\theta_2\epsilon_{t-2}$, resulting in equation 4.3.

$$y_t = \mu + \epsilon_t + \theta_1\epsilon_{t-1} + \theta_2\epsilon_{t-2}$$

Equation 4.3

Hence, we can see how the order q of the MA(q) process affects the number of past error terms that must be included in the model. The larger q is, the more past error terms affect the present value. Therefore, it is important to determine the order of the moving average process in order to fit the appropriate model, meaning that if we have a second-order

moving average process, then a second-order moving average model will be used for forecasting.

4.1.1 Identifying the order of a moving average process

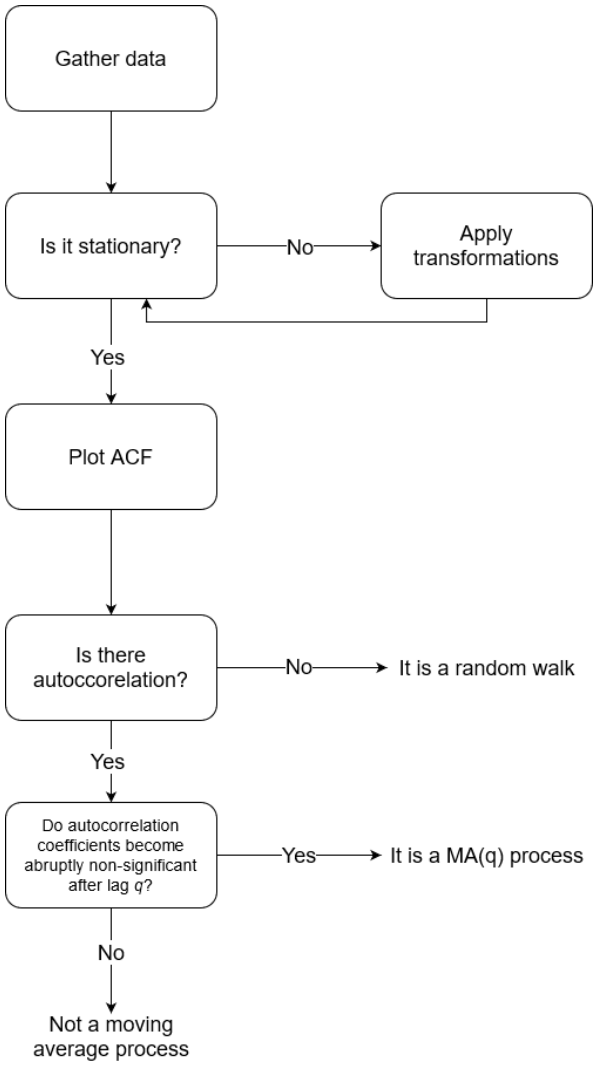


Figure 4.3. Steps to identify the order of a moving average process.

As usual, the first step is to gather the data. Then, we test for stationarity. In the event where our series is not stationary, we apply transformations, such as differencing, until the series is stationary. Then, we plot the ACF and look for significant autocorrelation coefficients. In the case of a random walk, we will not see significant coefficients after lag 0. On the other hand, if we see significant coefficients, then we must check if they become abruptly non-significant after some lag q. If that is the case, then we know that we have a moving average process of order q. Otherwise, we must follow a different set of steps to discover the underlying process of our time series.

Let's put this in action using our data for the volume of sales of widgets for the XYZ Widget Company. The dataset contains 500 days of sales volume data starting on January 1, 2019. We will follow the set of steps outlined in figure 4.3 and determine the order of the underlying moving average process.

The first step is to gather the data. While this step was already done for you, this is a great time to load the data into a `DataFrame` using `pandas` and display the first five rows of data:

```
import pandas as pd

df = pd.read_csv('data/widget_sales.csv')   #A
df.head()        #B
```

#A Read the CSV file into a DataFrame.
#B Display the first 5 rows of data

You see that our volume of sales is in the column *widget_sales*. Note that the volume of sales is in units of thousands of US dollars.

We can plot our data using matplotlib. Our values of interest are in the *widget_sales* columns so that is what we pass in to `ax.plot()`. Then, we give the x-axis the label of *Time*, and y-axis the label of *Widget sales (k$)*. Next, we specify the label for the ticks on the x-axis to display the month of the year. Finally, we tilt the x-ticks labels and remove extra whitespace around the figure using `plt.tight_layout()`. The result is figure 4.4.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

ax.plot(df.widget_sales)  #A
ax.set_xlabel('Time')     #B
ax.set_ylabel('Widget sales (k$)') #C

plt.xticks(
    [0, 30, 57, 87, 116, 145, 175, 204, 234, 264, 293, 323, 352, 382, 409, 439, 468, 498],
    ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec',
       '2020', 'Feb', 'Mar', 'Apr', 'May', 'Jun'])  #D

fig.autofmt_xdate()       #E
plt.tight_layout()        #F
```

#A Plot the volume of widget sales
#B Label the x-axis
#C Label the y-axis
#D Label the ticks on the x-axis

#E Tilt the labels on the x-ticks so that they display nicely
#F Remove extra whitespace around the figure

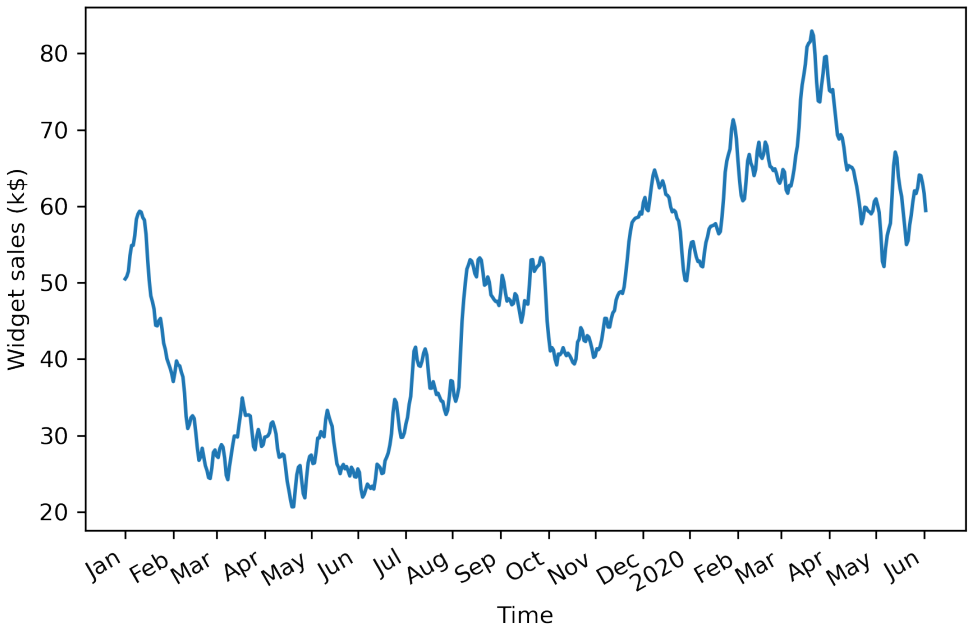


Figure 4.4 Volume of widget sales for the XYZ Widget Company over 500 days, starting in January 1, 2019.

The next step is to test for stationarity. We intuitively know that the series is not stationary since there is an observable trend as see in figure 4.4. Still, we will use the ADF test to make sure. Again, we use the `adfuller` function from the `statsmodels` library and extract the ADF statistic and p-value. If the ADF statistic is a large negative number and the p-value is smaller than 0.05, then our series is stationary. Otherwise, we must apply transformations.

```
from statsmodels.tsa.stattools import adfuller

ADF_result = adfuller(df.widget_sales)      #A

print(f'ADF Statistic: {ADF_result[0]}')    #B
print(f'p-value: {ADF_result[1]}') #C
```

#A Run the ADF test on the volume of widget sales, which is stored in the *widget_sales* column
#B Print the ADF statistic
#C Print the p-value

This results in an ADF statistic of -1.51 and a p-value of 0.53. Here, the ADF statistic is not a large negative number and the p-value is greater than 0.05. Therefore, our time series is not stationary and we must apply transformations to make it stationary.

In order to make our series stationary, we will try to stabilize the trend by applying a first-order differencing. We can do so by using the `diff` method from the `numpy` library. Remember that this method takes in a parameter `n` that specifies the order of differencing. In this case, because it is a first-order differencing, `n` will be equal to 1.

```
import numpy as np

widget_sales_diff = np.diff(df.widget_sales, n=1)    #A
```

#A Apply first-order differencing on our data and store the result in `widget_sales_diff`.

We can optionally plot the differenced series to see if we stabilized the trend. Figure 4.5 shows the differenced series. We can see that successfully removed long-term trend component of our series as values are hovering around 0 over the entire period.

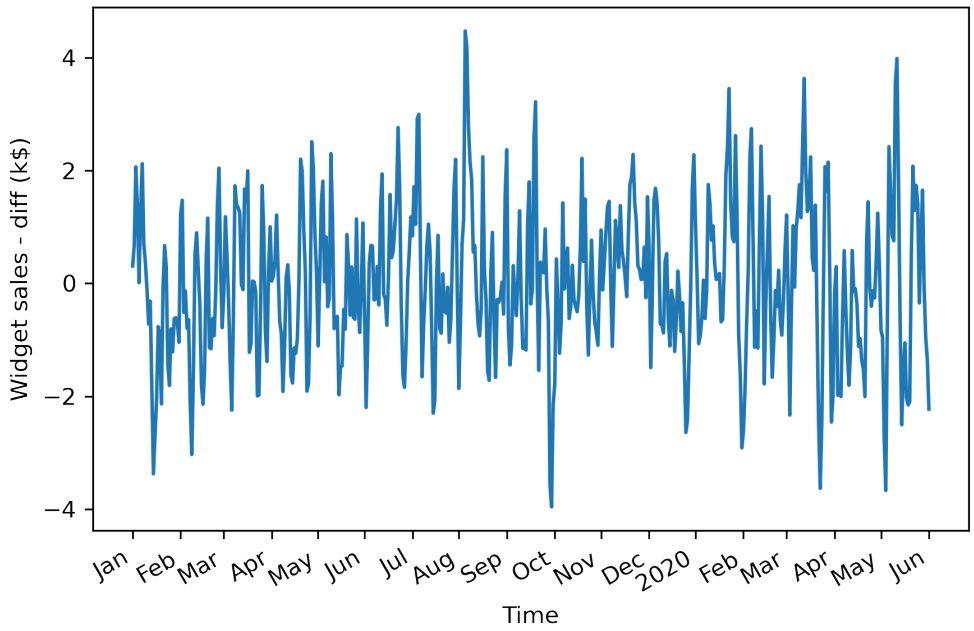


Figure 4.5. Differenced volume of widget sales. Notice how the trend component was stabilized since values

are hovering around 0 over our entire sample.

With a transformation applied to our series, we can test for stationarity again using the ADF test. This time, make sure to run the test on the differenced data stored in the `widget_sales_diff` variable.

```
ADF_result = adfuller(widget_sales_diff)    #A

print(f'ADF Statistic: {ADF_result[0]}')
print(f'p-value: {ADF_result[1]}')
```

#A Run the ADF test on the differenced time series

This gives an ADF statistic of -10.6 and a p-value of 7×10^{-19}. Therefore, with a large negative ADF statistic and a p-value much smaller than 0.05, we can say that our series is stationary.

Our next step is to plot the autocorrelation function. The `statsmodels` library conveniently includes the `plot_acf` function for us. We simply pass in our differenced series and specify the number of lags in the `lags` parameter. Remember that the number of lags determines the range of values on the x-axis.

```
from statsmodels.graphics.tsaplots import plot_acf

plot_acf(widget_sales_diff, lags=30);       #A

plt.tight_layout()
```

#A Plot the ACF of the differenced series

The resulting ACF plot is shown in figure 4.6. We notice that there are significant coefficients after lag 0. In fact, they are significant up until lag 2. Then, they abruptly become non-significant as they remain in the shaded area of the plot. We can see some significance around lag 20, but this is likely due to chance, as the following coefficients are not significant.

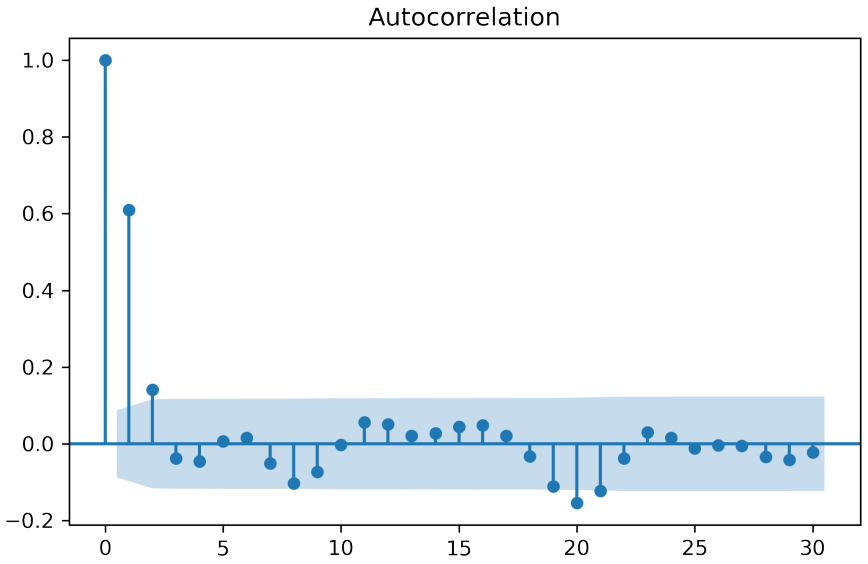


Figure 4.6 ACF plot of the differenced series. Notice how coefficients are significant up until lag 2 before falling abruptly in the non-significance zone (shaded area) of the plot. We see some significant coefficients around lag 20, but this is likely due to chance since they are non-significant between lag 3 and 20 and after lag 20.

Since we have significant autocorrelation coefficients up until lag 2, this means that we have a stationary moving average process of order 2. Therefore, we can use a second-order moving average model, or MA(2) model, to forecast our stationary time series.

Thus, we can see how the ACF plot helps us determine the order of a moving average process. The ACF plot will show significant autocorrelation coefficients up until lag q, after which all coefficients will be non-significant. We can then conclude that we have a moving average process of order q or MA(q) process. In our case, working with the volume of widget sales, we discovered that the stationary process is a second-order moving average process since the ACF plot showed significant coefficients up until lag 2.

4.2 Forecasting a moving average process

Once the order q of the moving average process is identified, we can fit the model to our training data and start forecasting. In our case, we discovered that the differenced volume of widget sales is a moving average process of order 2, or a MA(2) process.

The moving average model assumes stationarity, meaning that our forecasts must be done on a stationary time series. Therefore, we will train and test our model on the differenced volume of widget sales. We will try two naïve forecasting techniques and fit a second-order moving average model. The naïve forecast will serve as baseline to evaluate the performance of the moving average model, which we expect to be better than the

baselines since we previously identified our process to be a moving average process of order 2. Once we obtain our forecasts for the stationary process, we will have to *undifference* the forecasts, meaning that we must undo the process of differencing, in order to bring the forecasts back to their original scale.

In this scenario, we will allocate 90% of the data to the training set and reserve the other 10% for the test set meaning that we must forecast 50 timesteps into the future.

Let's split our data right away. We will first assign the differenced data to a `DataFrame`. Then, the training set will be the first 90% of the data, and the remaining 10% will be for the test set.

```
df_diff = pd.DataFrame({'widget_sales_diff': widget_sales_diff})      #A

train = df_diff[:int(0.9*len(df_diff))]     #B
test = df_diff[int(0.9*len(df_diff)):]      #C

print(len(train))
print(len(test))
```

#A Place the differenced data in a `DataFrame`
#B The first 90% of the data goes in the training set
#C Last 10% of the data goes in the test set for prediction

We print out size of the training set and test set to remind you of the data point that we lose when we difference. The original dataset contained 500 data points, while the differenced series contains a total of 499 data points since we differenced once.

Now, we can visualize the forecasting period for the differenced and undifferenced series. Here, we will make two subplots in the same figure. The result is shown in figure 4.7.

```
fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=1, sharex=True)#A

ax1.plot(df.widget_sales)
ax1.set_xlabel('Time')
ax1.set_ylabel('Widget sales (k$)')
ax1.axvspan(450, 500, color='#808080', alpha=0.2)

ax2.plot(df_diff.widget_sales_diff)
ax2.set_xlabel('Time')
ax2.set_ylabel('Value')
ax2.axvspan(450, 499, color='#808080', alpha=0.2)

plt.xticks(
    [0, 30, 57, 87, 116, 145, 175, 204, 234, 264, 293, 323, 352, 382, 409, 439, 468, 498],
    ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec',
       '2020', 'Feb', 'Mar', 'Apr', 'May', 'Jun'])

fig.autofmt_xdate()
plt.tight_layout()
```

#A Make 2 subplots inside the same figure.

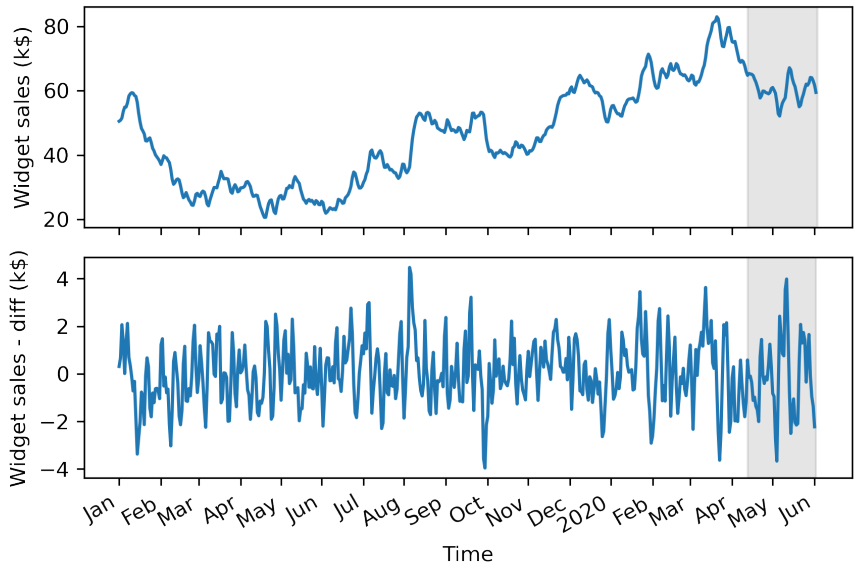


Figure 4.7 Forecasting period for the original and differenced series. Remember that our differenced series has one less data point than in its original state.

For the forecast horizon the moving average model bring in a particularity. The MA(q) model does not allow us to forecast 50 steps into future in one shot. Remember that the moving average model is linearly dependent on past error terms. Those terms are not observed in the dataset and must therefore be recursively estimated. This means that for an MA(q) model, we can only forecast q steps into the future. Any prediction made beyond that point will not have past error terms and the model will only predict the mean. Therefore, there is no added value in forecasting beyond q steps into the future because the predictions will fall flat, as only the mean is returned, which is equivalent to a baseline model.

In order to avoid simply predicting the mean beyond two timesteps into the future, we need to develop a function that will predict two timesteps or less at a time until 50 predictions are made, such that we can compare our predictions against the observed values of the test set. This method is called *rolling forecasts*. On the first pass, we will train on the 449 first timesteps and predict timesteps 450 and 451. Then, on the second pass, we will train on the 451 first timesteps, and predict timesteps 452 and 453. This is repeated until we finally predict the values at timestep 498 and 499.

Forecasting using the MA(q) model

When using a MA(q) model, forecasting beyond q steps into the future will simply return the mean, because there are no error terms to estimate beyond q steps. Therefore, we can do rolling forecasts to predict up to q steps at a time in order avoid predicting only the mean of the series.

We will compare our fitted MA(2) model to two baselines: the historical mean and the last value. That way, we can make sure that an MA(2) model will yield better predictions than naïve forecasts, which should be the case since we know the stationary process is a MA(2) process.

Note that you do not have to forecast 2-steps-ahead when you perform rolling forecasts with a MA(2) model. You can perform a 1-step ahead or to 2-step ahead forecasts repeatedly in order to avoid predicting only the mean. Similarly, with a MA(3) model, we could perform rolling forecasts with 1-step-ahed, 2-step ahead, or 3-step ahead rolling forecasts.

We need a function that will repeatedly fit a model and generate forecasts over a certain window of time, until forecasts for the entire test set are obtained. This function is shown in listing 4.1.

First, we will import the `SARIMAX` function from the `statsmodels` library. This function will allow us to fit a MA(2) model to our differenced series. Note that SARIMAX is a complex model that allows us to consider seasonal effects, autoregressive processes, non-stationary time series, moving average processes, and exogenous variables all in one single model. For now, we will disregard all factors except the moving average portion. We will gradually build upon the moving average model to eventually reach the SARIMAX model in later chapters.

Now, we define our `recursive_forecast` function. It will take in a `DataFrame`, the length of the training set, the forecast horizon, a window size, and a method.

The DataFrame contains the entire time series. The `train_len` parameter initializes the number of data points can be used to fit a model. As predictions are done, we can update this to simulate the observation of new values and use them to make the next sequence of forecasts. The `horizon` parameter is equal to length of the test set and represents how many values must be predicted. Then, the `window` parameter specifies how many timesteps are predicted at a time. In our case, because we have a MA(2) process, the window will be equal to 2. Finally, the `method` parameter specifies what model to use. The same function allows us to generate forecasts from the naïve methods and from the MA(2) model. Note the use of type hinting in the function declaration to prevent passing parameters of an unexpected type, hence avoiding our function to fail.

Then, each forecasting method is run in a loop. The loop starts at the **c** until `total_len` excluded, which is the sum of `train_len` and `horizon`, with steps of `window`. This loop generates a list of 25 values: $[449, 4451, 452,\ldots,497]$ but each pass generates two forecasts, thus returning a list of 50 forecasts for the entire test set.

Listing 4.1 A function for rolling forecasts on a horizon

```
from statsmodels.tsa.statespace.sarimax import SARIMAX

def rolling_forecast(df: pd.DataFrame, train_len: int, horizon: int, “[CA]”window: int,
       method: str) -> list:       #A

    total_len = train_len + horizon

    if method == 'mean':
```

```
        pred_mean = []

        for i in range(train_len, total_len, window):
            mean = np.mean(df[:i].values)
            pred_mean.extend(mean for _ in range(window))

        return pred_mean

    elif method == 'last':
        pred_last_value = []

        for i in range(train_len, total_len, window):
            last_value = df[:i].iloc[-1].values[0]
            pred_last_value.extend(last_value for _ in range(window))

        return pred_last_value

    elif method == 'MA':
        pred_MA = []

        for i in range(train_len, total_len, window):
            model = SARIMAX(df[:i], order=(0,0,2))  #B
            res = model.fit(disp=False)
            predictions = res.get_prediction(0, i + window - 1)
            oos_pred = predictions.predicted_mean.iloc[-window:]      #C
            pred_MA.extend(oos_pred)

        return pred_MA
```

#A The function takes in a `DataFrame` containing the full simulated moving average process. We also pass in the length of the training set (800 in our case) and the horizon of forecast (200 in our case). The next parameter specifies how many steps at time we wish to forecast (2 in our case). Finally, we specify the method to use to make forecasts.

#B The MA(q) model is part of the more complex model SARIMAX. We have not covered that yet, but we will slowly build upon the basic models to reach the SARIMAX model.

 # C The method `predicted_mean` is what allows us to retrieve the actual value of the forecast as defined by the `statsmodels` library.

Once defined, we can use our function and predict using the three following methods: predicting the historical mean, predicting the last value, and predicting using a fitted MA(2) model.

We first create a `DataFrame` to hold our predictions and we name it `pred_df`. We can copy the test set, such as to include the actual values in `pred_df`, making it easier to evaluate the performance of our models.

Then, we specify some constants. In Python, it is a good practice to name constants in capital letters. `TRAIN_LEN` is simply the length of our training set, the `HORIZON` is the length of the test set, which is 50 days, and the `WINDOW` can be 1 or 2 because we are using a MA(2) model, and predicting beyond two steps into future will simply return the mean. In this case, we will use a value of 2.

Next, we use our `rolling_forecast` function to generate a list of prediction for each method. Each list of predictions is then stored in its own column in `pred_df`.

```
pred_df = test.copy()
```

```
TRAIN_LEN = len(train)
HORIZON = len(test)
WINDOW = 2

pred_mean = rolling_forecast(df_diff, TRAIN_LEN, HORIZON, WINDOW, 'mean')
pred_last_value = rolling_forecast(df_diff, TRAIN_LEN, HORIZON, WINDOW, “[CA]”'last')
pred_MA = rolling_forecast(df_diff, TRAIN_LEN, HORIZON, WINDOW, 'MA')

pred_df['pred_mean'] = pred_mean
pred_df['pred_last_value'] = pred_last_value
pred_df['pred_MA'] = pred_MA
```

We can visualize how predictions against the observed values in the test set. Keep in mind that we are still working with the differenced dataset, so our predictions are also differenced values.

For this figure, we will plot part of the training data, to see the transition between the training set and the test set. Then, our observed values will be a solid line, and we will label this curve as *actual*. Then, we plot the forecasts coming from the historical mean, the forecasts coming from the last observed value, and the forecasts coming from the MA(2) model. They will respectively be a dotted line, a dotted and dashed line, and a black dashed line, with labels of *mean*, *last* and *MA(2)*. The result is shown in figure 4.8.

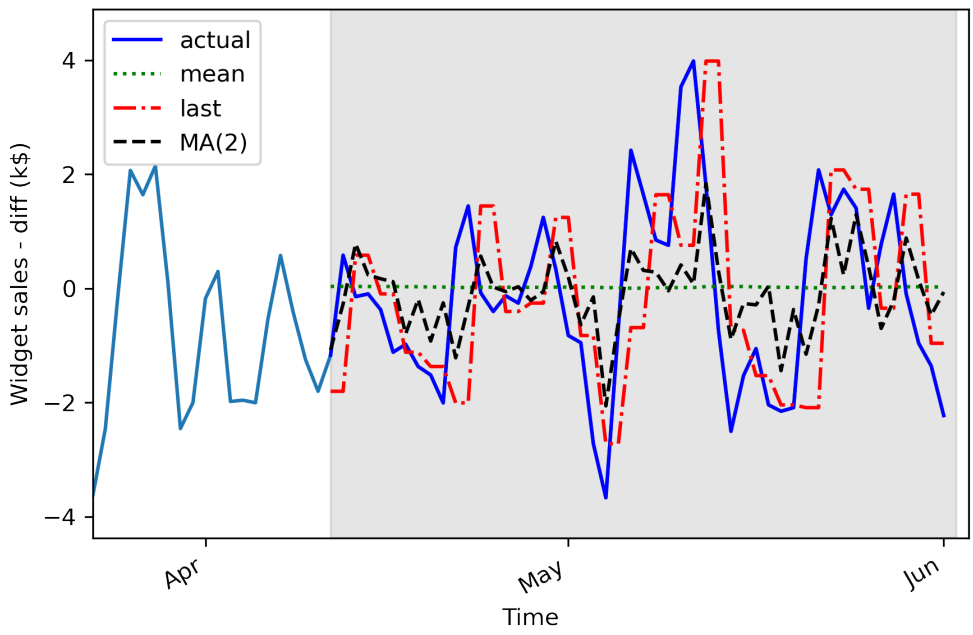


Figure 4.8 Forecasts of the differenced volume of widget sales. In a professional setting, it does not make

sense to report differenced predictions. Therefore, we will undo the transformation later on.

From figure 4.8, we notice that the prediction coming from the historical mean, shown as a dotted line, is almost a straight line. This is expected since the process is stationary and so the historical mean should be stable over time.

The next step is to measure the performance of our models. To do so, we will calculate the mean squared error or MSE. Here, we will use the `mean_squared_error` function from the `sklearn` package. We simply need to pass the observed values and the predicted values into the function.

```
from sklearn.metrics import mean_squared_error

mse_mean = mean_squared_error(pred_df.widget_sales_diff, pred_df.pred_mean)
mse_last = mean_squared_error(pred_df.widget_sales_diff, pred_df.pred_last_value)
```

```
print(mse_mean, mse_last, mse_MA)
```

This prints out an MSE of 2.56 for the historical mean method, 3.25 for the last value method, and 1.95 for the MA(2) model. Here, our MA(2) model is the best performing forecasting method since its MSE is the lowest of the three methods. This is expected because we previously identified a second-order moving average process for the differenced volume of widget sales, thus resulting in a smaller MSE compared to the naïve forecasting methods. We can visualize the MSE for all forecasting techniques in figure 4.9.

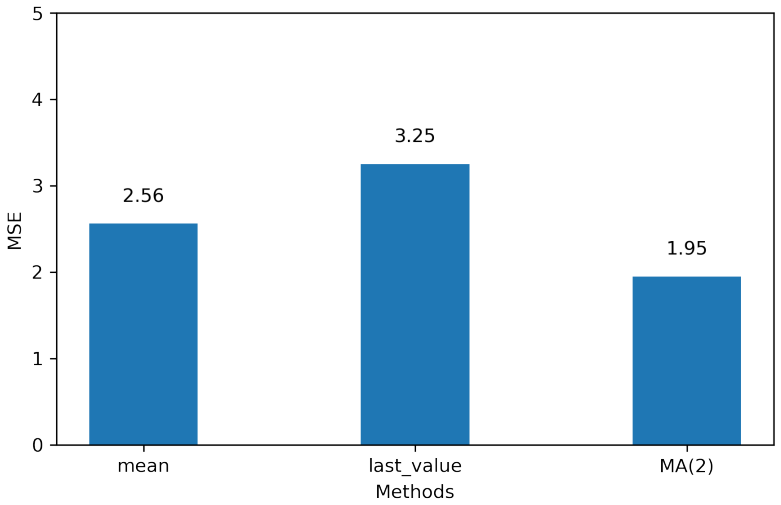


Figure 4.9 MSE for each forecasting method on the difference volume of widget sales. Here, the MA(2) model is the champion model, since its MSE is the lowest.

Now that we have our champion model on the stationary series, we need to *undifference* our predictions to bring them back to the original scale of the untransformed dataset. Recall that differencing is the result of the difference between a value at time t and its previous value as shown in figure 4.10.

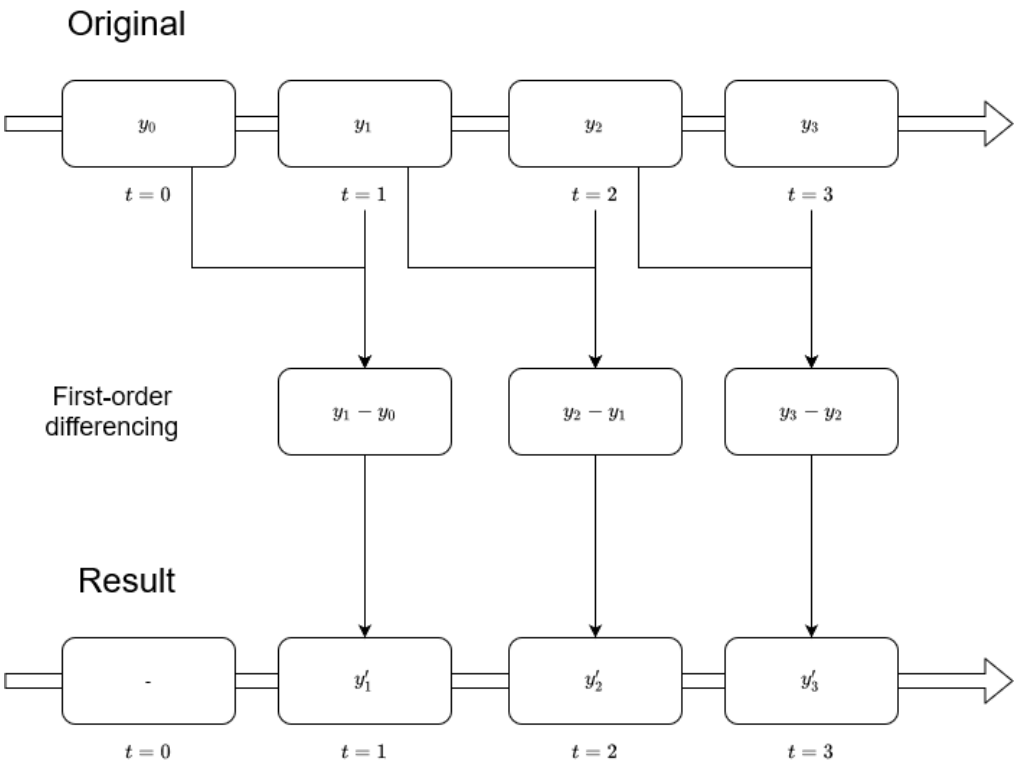


Figure 4.10 Visualizing a first-order difference

In order to reverse our first-order difference, we need to add an initial value y_0 to the first differenced value y'_1. That way, we can recover y_1 in its original scale. This is what is demonstrated in equation 4.4

$$y_1 = y_0 + y'_1 = y_0 + y_1 - y_0 = y_1$$

Equation 4.4

Then, y_2 can be obtained using a cumulative sum of the differenced values, as shown in equation 4.5.

$$y_2 = y_0 + y'_1 + y'_2 = y_0 + y_1 - y_0 + y_2 - y_1 = (y_0 - y_0) + (y_1 - y_1) + y_2 = y_2$$

Therefore, applying the cumulative sum once will undo a first-order differencing. In the case where the series was differenced twice to become stationary, then we need to repeat this process twice.

Thus, to obtain our predictions in the original scale of our dataset, we need to use the first value of the test as our initial value. Then we will perform a cumulative sum to obtain a series of 50 predictions in the original scale of the dataset. We will assign these predictions in the *pred_widget_sales* column.

```
df['pred_widget_sales'] = pd.Series()       #A
df['pred_widget_sales'][450:] = df['widget_sales'].iloc[450] + pred_df['pred_MA'].cumsum()
         #B
```

#A Initialize an empty column to hold our predictions
#B *Undifference* the predictions to bring them back to the original scale of the dataset.

Let's visualize our untransformed predictions against the recorded data. The result is shown in figure 4.11. Remember that we are now using the original dataset stored in `df`.

```
fig, ax = plt.subplots()

ax.plot(df.widget_sales)
ax.plot(df.widget_sales, 'b-', label='actual')      #A
ax.plot(df.pred_widget_sales, 'k--', label='MA(2)') #B

ax.legend(loc=2)

ax.set_xlabel('Time')
ax.set_ylabel('Widget sales (K4)')

ax.axvspan(450, 500, color='#808080', alpha=0.2)

ax.set_xlim(400, 500)

plt.xticks(
    [409, 439, 468, 498],
    ['Mar', 'Apr', 'May', 'Jun'])

fig.autofmt_xdate()
plt.tight_layout()
```

#A Plot the actual values
#B Plot the undifferenced predictions

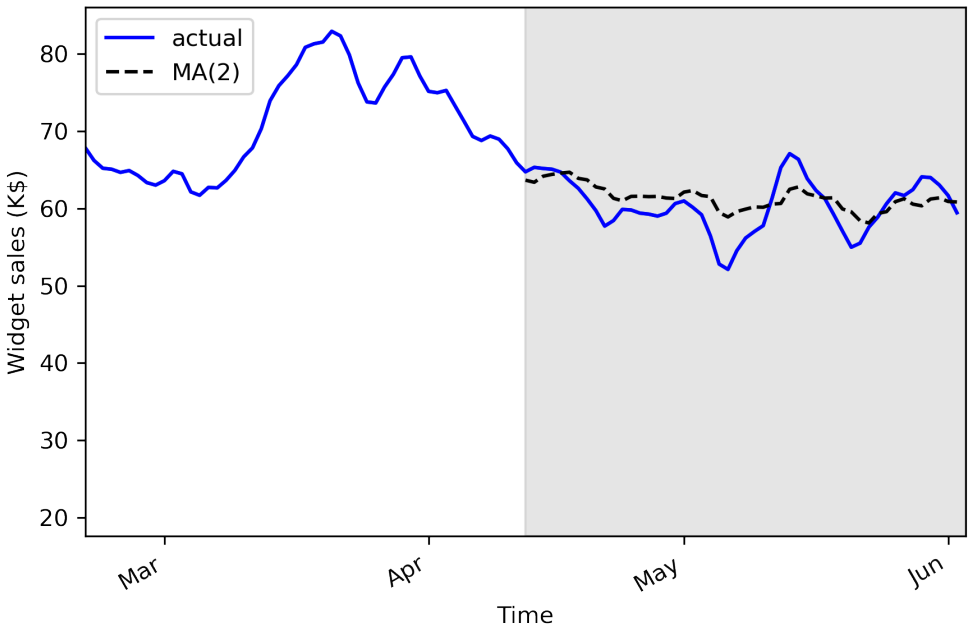


Figure 4.11 Undifferenced MA(2) forecasts.

We can see in figure 4.11 that our forecast curve, shown with a dashed line, follows the general trend of the observed values, although it does not predict bigger troughs and peaks.

The final step is to report the MSE on the original dataset. In a professional setting, we would not report the differenced predictions, because they do not make sense from a business perspective; we must report values and errors in the original scale of the data.

Now, we will measure the mean absolute erorr, or MAE, using the `mean_absolute_error` function from `sklearn`. We use this metric because it easy to interpret, as it returns the average of the absolute difference between the predicted and actual values, instead of a squared difference like the MSE.

```
from sklearn.metrics import mean_absolute_error

mae_MA_undiff = mean_absolute_error(df.widget_sales[450:], df.pred_widget_sales[450:])

print(mae_MA_undiff)
```

This prints out a MAE of 2.32. Therefore, our predictions are, on average, off by 2320$, either above or below the actual value. Remember that our data has units of thousands of dollars, so we multiply the MAE by 1000 to express the average absolute difference.

4.3 Next steps

In this chapter, we covered the moving average process and how it can be modeled by an MA(q) model where q is the order. We learned that to identify a moving average process, we must study the ACF plot once it is stationary. Then, the ACF plot will show significant peaks all the way to lag q, and the rest will not be significantly different from 0.

However, it is possible that when studying the ACF plot of a stationary process, we see a sinusoidal pattern, with negative coefficients and significant autocorrelation at large lags. For now, as shown in figure 4.12, we simply know that it is not a moving average process.

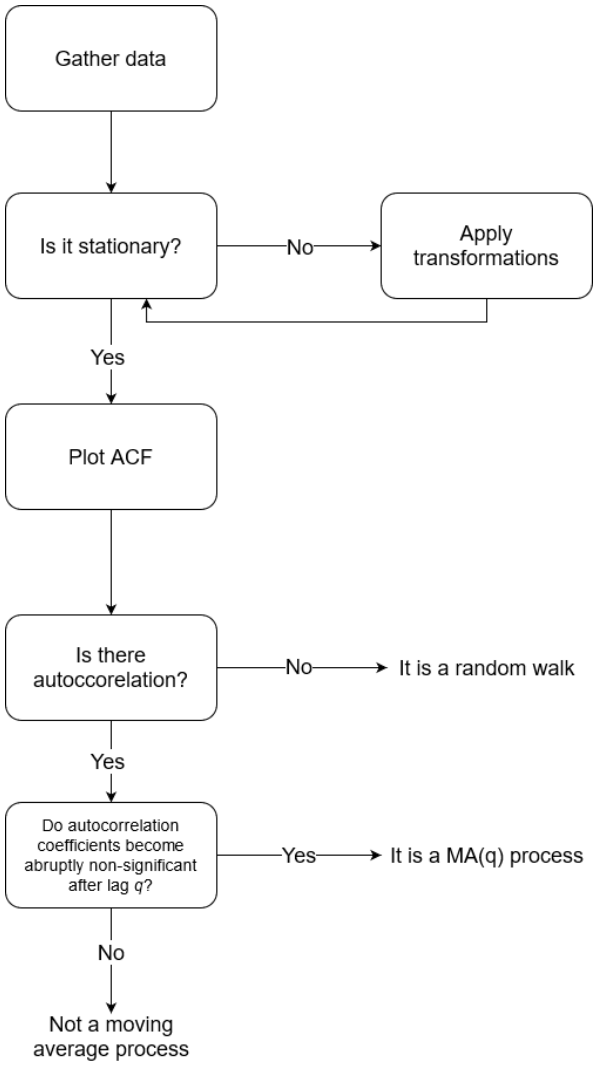


Figure 4.12 Steps to identify the underlying process of a stationary time series.

In a case where we see a sinusoidal pattern in the ACF plot of a stationary process, it is a hint that an autoregressive process is at play and we must use an AR(p) model to produce our forecast. Just like the MA(q) model, the AR(p) model will require us to identify its order. This time we will have to plot the **partial** autocorrelation function and see at which lag coefficients suddenly become non-significant.

The next chapter will focus entirely on the autoregressive process, how to identify its order, and how to forecast such a process.

4.4 Exercises

4.4.1 Easy: Simulate a MA(2) process and make forecasts

- Simulate a stationary MA(2) process. To do so, use the `ArmaProcess` function from the `statsmodels` library and simulate the following process:
- $y_t = 0.9\theta_{t-1} + 0.3\theta_{t-1}$
- For this exercise, generate 1000 samples.

```
from statsmodels.tsa.arima_process import ArmaProcess
import numpy as np

np.random.seed(42)        #A

ma2 = np.array([1, 0.9, 0.3])
ar2 = np.array([1, 0, 0])

MA2_process = ArmaProcess(ar2, ma2).generate_sample(nsample=1000)
```

#A Set the seed for reproducibility. There is randomness every time you generate samples. By setting the seed, you make sure that the same values are obtained no matter how many times the code is run. Change the seed if you want to experiment with different values.

- Plot your simulated moving average.
- Run the ADF test and check if the process is stationary.
- Plot the ACF and see if there are or are not significant coefficients after lag 2.
- Separate your simulates series into a training and test set. Take the first 800 timesteps for the training set and assign the rest to the test set.
- Make forecasts over the test set. Use the mean, last value, and a MA(2) model. Make sure to repeatedly forecast 2 timesteps at time using the `recursive_forecast` function as defined.
- Plot your forecasts.
- Measure the MSE and identify your champion model.
- Plot your MSEs in a bar plot.

4.4.2 Medium: Simulate a MA(q) process and make forecasts

- Recreate the previous exercise but simulate a moving average process of your choice. Try simulating a third-order or fourth-order moving average process. Here, I would recommend generating 10,000 samples. Be especially attentive to the ACF and see if your coefficients become non-significant after lag q.

4.5 Summary

- A moving average process states that the present value is linearly dependent on the mean, present error term, and past error terms. The error terms are normally distributed.
- We can identify the order q of a stationary moving average process by studying the ACF plot. The coefficients are significant up until lag q only.
- We can predict up to q steps into the future because the error terms are not observed

in the data and must be recursively estimated.

- Predicting beyond q steps into the future will simply return the mean of the series. To avoid that, we apply rolling forecasts.
- If we apply a transformation to the data, we must undo it to bring our predictions back to the original scale of the data.
- The moving average model assumes the data is stationary. Therefore, we can only use this model on stationary data.

5

Modeling an autoregressive process

This chapter covers

- **Defining an autoregressive process**
- **Defining the PACF**
- **Using the PACF plot to determine the order of an autoregressive process**
- **Forecasting a time series using the autoregressive model**

In the previous chapter, we covered the moving average process, also denoted as MA(q), where *q* is the order. We learned that in a moving average process, the present value is linearly dependent on current and past error terms. Therefore, if we predict more than *q* steps ahead, the prediction will fall flat and will return only the mean of the series because the error terms are not observed in the data and must be recursively estimated. Finally, we saw that we can determine the order of a stationary MA(q) process by studying the ACF plot; the autocorrelation coefficients will be significant up until lag *q*. In the case where the autocorrelation coefficients slowly decay or exhibit a sinusoidal pattern, then we are possibly in the presence of an autoregressive process.

In this chapter, we will first define the autoregressive process. Then, we will define the partial autocorrelation function and use it to find the order of the underlying autoregressive process of our dataset. Finally, we will use the AR(p) model to produce forecasts.

5.1 Predicting the average weekly foot traffic in a retail store

Suppose that you want to forecast the average weekly foot traffic in a retail store so that the store director can better manage the schedule staff's schedule. If many people are expected to come to the store, then more employees should be present to provide assistance. Of course, if less people are expected to visit the store then the director can schedule fewer employees to work during that week so that the store can optimize its spending on salary and ensure that employees are not overwhelmed or underwhelmed by store visitors.

In this situation, we collected 1000 datapoints, each representing the average weekly foot traffic at a retail store starting in 2000. We can see the evolution of our data through time in figure 5.1.

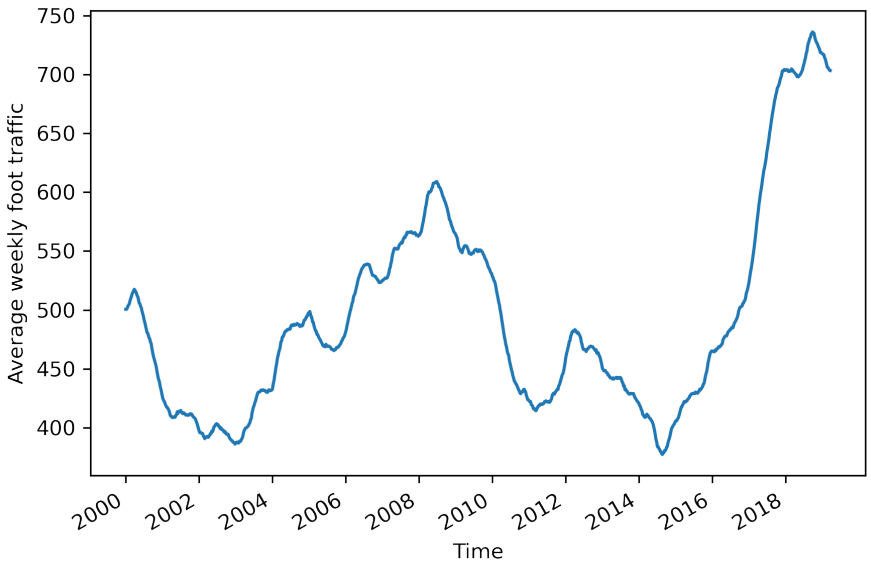


Figure 5.1 Average weekly foot traffic in a retail store. The dataset contains 1000 datapoints, starting in the first week of 2000. Note that this is fictional data.

Looking at figure 5.1, we can see a long-term trend with peaks and troughs along the way. From this observation, we can intuitively say that this time series is not a stationary process since we observe a trend over time. Furthermore, there is no apparent cyclical pattern in the data so we can rule out any seasonal effects for now.

Again, in order to forecast the average weekly foot traffic, we need to identify the underlying process. Thus, we must apply the same steps that we covered in chapter 4. That way, we can verify whether we have a random walk or a moving average process at play. The steps are shown in figure 5.2.

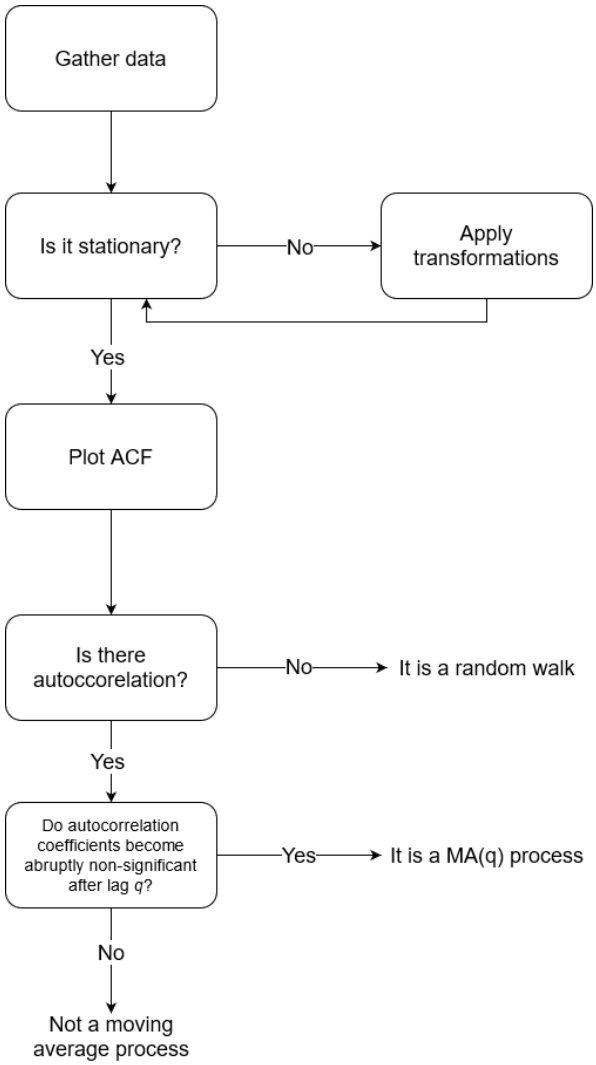


Figure 5.2 Steps to identify the underlying process of a stationary time series. For now, we can identify a random walk or a moving average process.

Here the data is already collected, so we can move on to testing for stationarity. As aforementioned, the presence of a trend over time means that our series is likely not stationary, so we will have to apply a transformation in order to make it stationary. Then, we will plot the ACF. As we work through the chapter, we will notice that not only is there autocorrelation, but the ACF plot will have a slowly decaying trend.

This is indicative of an autoregressive process or order p, also denoted as AR(p). In this case, we must plot the **partial autocorrelation function** or PACF to find the order p. Just like the coefficients on a ACF plot for a MA(q) process, the coefficients on the PACF plot will become abruptly non-significant after lag p, hence determining the order of the autoregressive process.

Again, the order of the autoregressive process determines how many parameters must be included in the AR(p) model. Then, we are ready to make forecasts. In this situation, we wish to forecast next week's average foot traffic.

5.2 Defining the autoregressive process

An autoregressive process establishes that the output variable depends linearly on its own previous values. In other words, it is a regression of the variable against itself.

An autoregressive process is denoted as an AR(p) process, where *p* is the order. In such process, the present value y_t is a linear combination of a constant C, the present error term ϵ_t which is also white noise, and the past values of the series $y_{t-\mathrm{p}}$. The magnitude of the influence of the past values on the present value is denoted as ϕ_p, which represent the coefficients of the AR(p) model. Mathematically, we express a general AR(p) model as equation 5.1.

$$y_t = C + \phi_1 y_{t-1} + \epsilon_t + \phi_2 y_{t-2} + \epsilon_t + \cdots + \phi_\mathrm{p} y_{t-\mathrm{p}} + \epsilon_t$$

Equation 5.1

Autoregressive process

An autoregressive process is a regression of a variable against itself. In time series, this means that the present value is linearly dependent on its past values.

The autoregressive process is denoted as AR(p) where p is the order. The general expression of an AR(p) model is:

$$y_t = C + \phi_1 y_{t-1} + \epsilon_t + \phi_2 y_{t-2} + \epsilon_t + \cdots + \phi_\mathrm{p} y_{t-\mathrm{p}} + \epsilon_t$$

Similar to the moving average process, the order p of an autoregressive process determines the number of past values that affect the present value. If we have a first-order autoregressive process, also denoted as AR(1), then the present value y_t is only dependent on a constant C, the value at the previous timestep $\phi_1 y_{t-1}$, and some white noise ϵ_t, as shown in equation 5.2.

$$y_t = C + \phi_1 y_{t-1} + \epsilon_t$$

Equation 5.2

Looking at equation 5.2, you might notice that it is very similar to a random walk process, which we covered in chapter 3. In fact, if $\phi_1 = 1$, then equation 5.2 becomes:

$$y_t = C + y_{t-1} + \epsilon_t$$

Which is our random walk model. Therefore, we can say that the random walk is a special case of an autoregressive process, where the order p is 1 and ϕ_1 is equal to 1 as well. Notice also that if C is not equal to 0, then we have a random walk with drift.

In the case where we have a second-order autoregressive process, or AR(2), then the present value y_t is linearly dependent on a constant C, the value at the previous timestep $\phi_1 y_{t-1}$, the value two timesteps prior $\phi_2 y_{t-2}$, and the present error term ϵ_t, as shown in equation 5.3.

$$y_t = C + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \epsilon_t$$

Equation 5.3

We see how the order p influences the number of parameters that must be included in our model. As with a moving average process, we must find the right order of an autoregressive process in order to build the appropriate model. This means that if we identify an AR(3) process, then we will use a third-order autoregressive model to make forecasts.

5.3 Finding the order of a stationary autoregressive process

Again, just like with the moving average process, there is a way to determine the order p of a stationary autoregressive process. Here, we extend the steps needed to identify the order of a moving average as shown in figure 5.3.

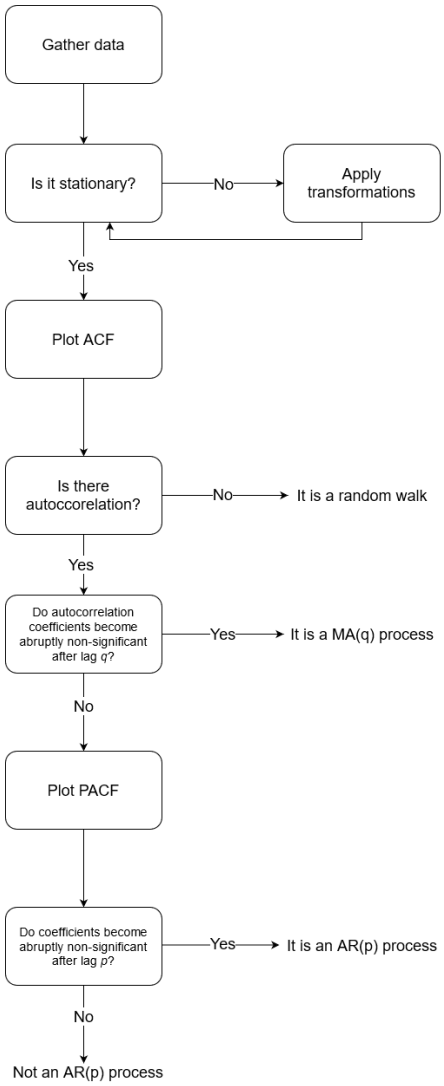


Figure 5.3 Steps to identify the order of an autoregressive process.

The natural first step is to collect the data. Here, we will work with the average weekly foot traffic dataset that we introduced at the beginning of the chapter. We will read the data using `pandas` and store it as a `DataFrame`.

```
import pandas as pd

df = pd.read_csv('data/foot_traffic.csv')   #A
```

```
df.head()        #B
```

#A Read the CSV file into a `DataFrame`
#B Display the first 5 rows of data

We see that our data contains a single column *foot_traffic*, in which the average weekly foot traffic at the retail store is recorded.

As always, we will plot our data to see if there are any observable patterns, such as a trend or seasonality. By now, you should be comfortable with plotting time series, so we will not dive deeply in the code to generate the graph. The result is the plot shown in figure 5.4.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

ax.plot(df.foot_traffic)  #A
ax.set_xlabel('Time')     #B
ax.set_ylabel('Average weekly foot traffic')        #C

plt.xticks(np.arange(0, 1000, 104), np.arange(2000, 2020, 2))#D

fig.autofmt_xdate()       #E
plt.tight_layout()        #F
```

#A Plot the average weekly foot traffic at a retail store
#B Label the x-axis
#C Label the y-axis
#D Label the ticks on the x-axis
#E Tilt the labels on the x-ticks so that they display nicely
#F Remove extra whitespace around the figure

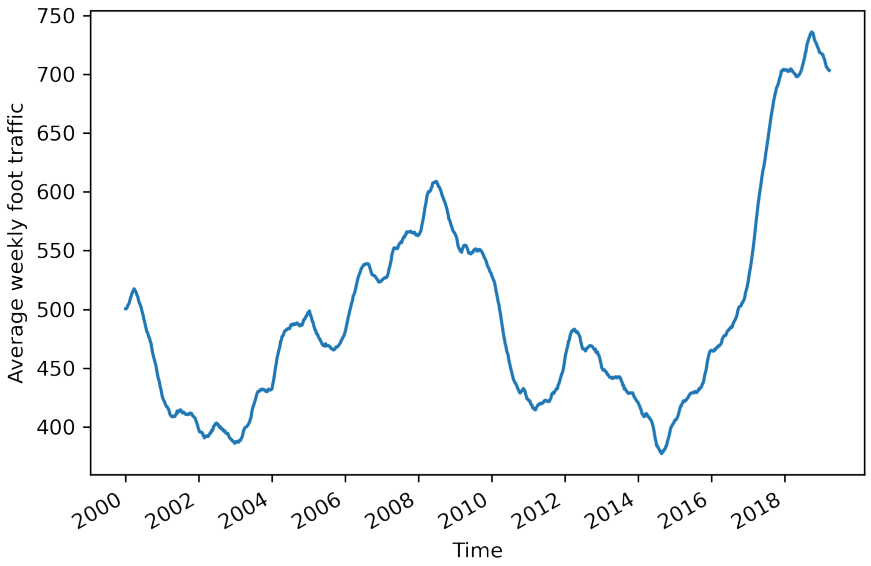


Figure 5.4 Average weekly foot traffic in a retail store. The dataset contains 1000 datapoints, starting in the first week of 2000.

Looking at figure 5.4, we notice that there is no cyclical pattern so we can rule out the presence the seasonality. As for the trend, we notice that it is sometimes positive, sometimes negative throughout the years, with the most recent trend being positive since 2016.

The next step is then to check for stationarity. As mentioned before, the presence of a trend means that our series is like non-stationary. Let's verify that using the ADF test. Again, you should be comfortable running without detailed explanation of the code.

```
from statsmodels.tsa.stattools import adfuller

ADF_result = adfuller(df.foot_traffic)      #A

print(f'ADF Statistic: {ADF_result[0]}')    #B
print(f'p-value: {ADF_result[1]}') #C
```

#A Run the ADF test on the average weekly foot traffic, which is stored in the *foot_traffic* column
#B Print the ADF statistic
#C Print the p-value

This prints out an ADF statistic of -1.18 along with a p-value of 0.68. Since the ADF statistic is not a large negative number, and with a p-value greater than 0.05, we cannot reject the null hypothesis and our series is therefore non-stationary.

Hence, we must apply a transformation in order to make it stationary. To remove the effect of the trend and stabilize the mean of the series, we will use differencing.

```
import numpy as np

foot_traffic_diff = np.diff(df.foot_traffic, n=1)    #A
```

#A Apply a first-order differencing on our data and store the result in `foot_traffic_diff`

Optionally, we can plot our differenced series `foot_traffic_diff` to see if we successfully removed the effect of the trend. The differenced series is shown in figure 5.5. We can see that we indeed removed the long-term trend, since the series starts and finished roughly at the same value.

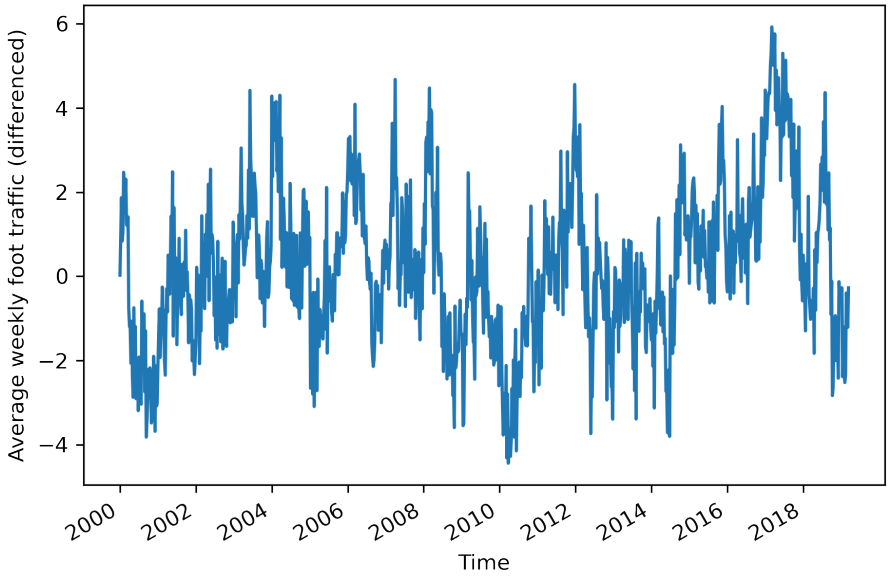


Figure 5.5 Differenced average weekly foot traffic at a retail store. Notice how the trend effect was removed, since the series starts and ends at roughly the same value.

> **CAN YOU RECREATE FIGURE 5.5?** While optional, it is a good idea to plot your series as you apply transformations. This gives a better intuition as to whether a series is stationary or not after a particular transformation. Try recreating figure 5.5 on your own.

With a transformation applied to the series, we can verify if the series is stationary by running the ADF test on the differenced series.

```
ADF_result = adfuller(foot_traffic_diff)    #A

print(f'ADF Statistic: {ADF_result[0]}')
print(f'p-value: {ADF_result[1]}')
```

```
#A Run the ADF test on the differenced time series
```

This prints out an ADF statistic of -5.27 and a p-value of 6.36×10^{-6}. With a p-value smaller than 0.05, we can reject the null hypothesis, meaning that we have now a stationary series.

The next step is to plot the ACF and see if there is autocorrelation and if the coefficients become abruptly non-significant after a certain lag. As we did in the two previous chapters, we will use the `plot_acf` function from `statsmodels`. The result is shown in figure 5.6.

```
from statsmodels.graphics.tsaplots import plot_acf

plot_acf(foot_traffic_diff, lags=20);       #A

plt.tight_layout()
```

```
#A Plot the ACF of the differenced series
```

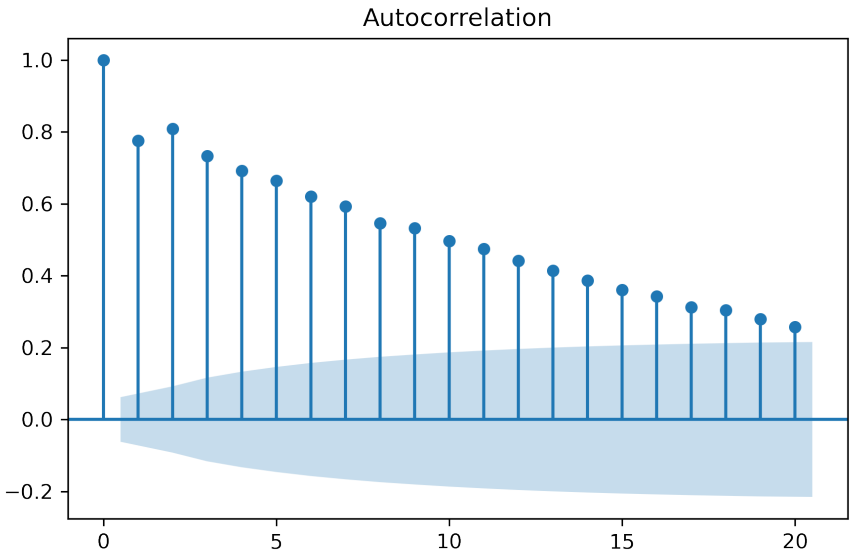


Figure 5.6 ACF plot of the differenced average weekly foot traffic at a retail store. Notice how the plot is slowly decaying. This is a behavior that we have not observed until now, and it is indicative of an autoregressive process.

Looking at figure 5.6, we notice that we have significant autocorrelation coefficients beyond lag 0. Therefore, we know that our process is not a random walk. Furthermore, we notice that the coefficients are slowly decaying as the lag increases. Therefore, there is not lag at which the coefficients abruptly become non-significant. This means that we do not have a moving average process and that we are likely studying an autoregressive process.

When the ACF plot of a stationary process exhibits a slowly decaying pattern, then it is a hint that we probably have an autoregressive process in play. We must then find another way to identify the order p of our AR(p) process. Specifically, we must turn our attention to the **partial autocorrelation function** plot or **PACF.**

5.3.1 The partial autocorrelation function (PACF)

In an attempt to identify the order of a stationary autoregressive process, we used the ACF plot just like we did for a moving average process. Unfortunately, the ACF plot cannot give us this information and therefore we must turn to the **partial autocorrelation function** or **PACF**.

Remember that the autocorrelation measures the linear relationship between lagged values of a time series. Consequently, the autocorrelation function measures how the correlation changes between two values as the lag is increased.

To understand the partial autocorrelation function, or PACF, let's consider the following scenario. Suppose that we have the following AR(2) process:

$$y_t = 0.33y_{t-1} + 0.50y_{t-2}$$

Equation 5.4

Now, we wish to measure how y_t relates to y_{t-1}, or in other words, we want to measure their correlation. This is done by the autocorrelation function (ACF). However, from the equation, we can see that y_{t-2} also has an influence on y_t. Even more important, it also has an impact on the value of y_{t-1}, since by the nature of an AR(2) process, each value depends on the previous two values. Therefore, when we measure the autocorrelation between y_t and y_{t-1} using the ACF, we are not taking into account the fact that y_{t-2} also has an influence on both y_t and y_{t-1}. This means that we are not measuring the *true* impact of y_{t-1} on y_t. To do so, we must remove the effect of y_{t-2}. That way, we are effectively measuring the real association between y_t and y_{t-1}. Thus, we are measuring the partial autocorrelation between y_t and y_{t-1}.

In more formal terms, the partial autocorrelation measures the correlation between lagged values in a time series when we remove the influence of other correlated lagged values. Those are known as confounding variables. Of course, the partial autocorrelation function will reveal how the partial autocorrelation varies when the lag increases.

Partial autocorrelation

Partial autocorrelation measures the correlation between lagged values in a time series when we remove the influence of other correlated lagged values. We can plot the partial autocorrelation function to determine the order of a stationary AR(p) process. The coefficients will be non-significant after lag p.

Let's verify if plotting the PACF will reveal the order of our process shown in equation 5.4. We know from equation 5.4 that we have a second-order autoregressive process or AR(2). We will simulate it using the `ArmaProcess` function from `statsmodels`. The function expects an array containing the coefficients of an MA(q) process and an array containing the coefficients for an AR(p) process. Since we are only interested in simulating an AR(2) process, we will set the coefficients of the MA(q) process to 0. Then, as specified by the `statsmodels` documentation, the coefficients of the AR(2) process must have opposite signs to that we wish to simulate. Therefore, the array will contain -0.33, and -0.50. In addition, the function requires us to include the coefficient at lag 0, which is the number that multiplies y_t. Here, that number is simply 1.

Once the arrays of coefficients are defined, we can feed them to the `ArmaProcess` function, and we will generate 1000 samples. Make sure to set the random seed to 42 in order to reproduce the results shown here.

```
from statsmodels.tsa.arima_process import ArmaProcess
import numpy as np

np.random.seed(42)        #A

ma2 = np.array([1, 0, 0]) #B
ar2 = np.array([1, -0.33, -0.50])  #C

AR2_process = ArmaProcess(ar2, ma2).generate_sample(nsample=1000)     #D
```

#A Set the random seed to 42 in order to reproduce the results shown here
#B Set the coefficients of the MA(q) process to 0, since we are only interested in simulating an AR(2) process. Note that the first coefficient is 1 for the 0th lag, and it must be provided as specified by the documentation.
#C Set the coefficients for the AR(2) process. Again, the coefficients at the 0th lag is 1. Then, we must write the coefficients with opposite signs to what we defined in equation 5.4, as specified by the documentation.
#D Simulate the AR(2) process and generate 1000 samples.

Now that we have a simulated AR(2) process, let's plot the PACF and see if the coefficients become abruptly non-significant after lag 2. If that is the case, then we know that we can use the PACF plot to determine the order of a stationary autoregressive process, just like we can use the ACF plot to determine the order of a stationary moving average process.

The `statsmodels` library also allows us to plot the PACF rapidly. We can use the `plot_pacf` function, which simply requires our series and the number of lags to display on the plot. The resulting plot is shown in figure 5.7.

```
from statsmodels.graphics.tsaplots import plot_pacf

plot_pacf(AR2_process, lags=20);   #A

plt.tight_layout()
```

#A Plot the PACF of our simulated AR(2) process

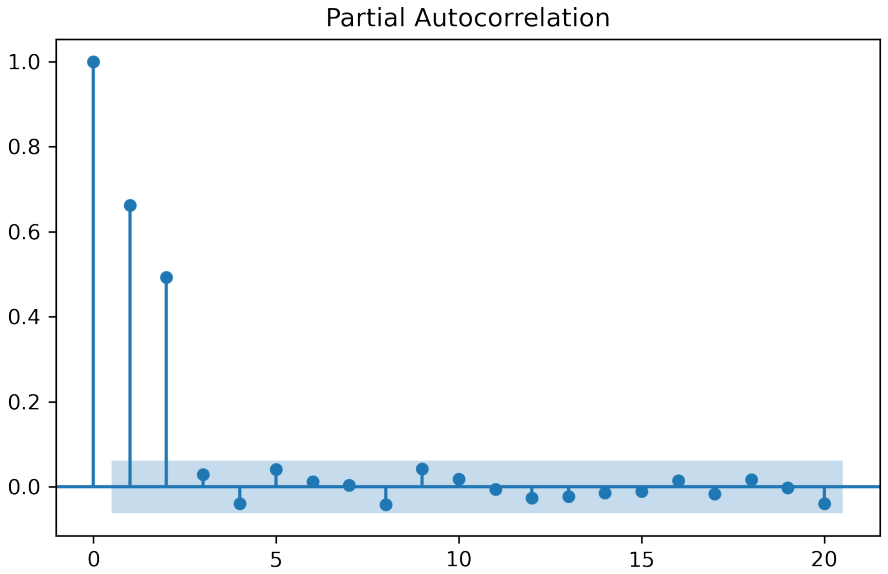


Figure 5.7 Plot of the PACF for our simulated AR(2) process. We can clearly see here that after lag 2, the partial autocorrelation coefficients are not significantly different from 0. Therefore, we can identify the order of a stationary AR(p) model using the PACF plot.

In figure 5.7, we can see how after lag 2, the partial autocorrelation coefficients are not significantly different from 0, and we therefore conclude that we have an autoregressive process of order 2.

We now know that we can use the PACF plot to identify the order of a stationary AR(p) process. The coefficients in the PACF plot will be significant up until lag *p*. Afterwards, they should not be significantly different from 0.

Let's see if we can apply the same strategy on our average weekly foot traffic dataset. We made the series stationary and saw that the ACF plot exhibited a slowly decaying trend. Now, we will plot the PACF to see if we the lags become non-significant after a particular lag.

The process is exactly the same as what we just did, only this time we will plot the PACF of our differenced series stored in `foot_traffic_diff`. We can see the resulting plot in figure 5.8.

```
plot_pacf(foot_traffic_diff, lags=20);      #A

plt.tight_layout()
```

#A Plot the PACF of our differenced series

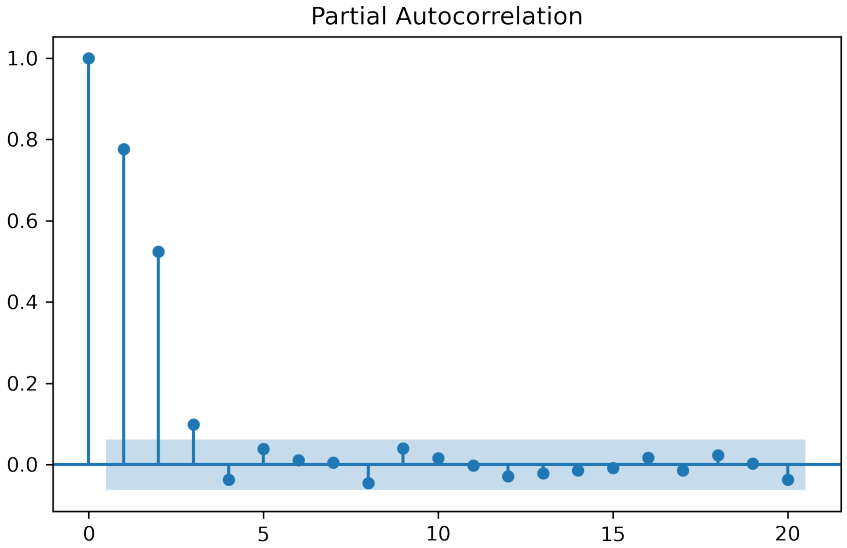


Figure 5.8 PACF of our differenced average weekly foot traffic in a retail store. We can show the coefficients are non-significant after lag 3. Therefore, we can say that our stationary process is a third-order autoregressive process or an AR(3) process.

Looking at figure 5.8, we see that there are not significant coefficients after lag 3. Therefore, we can say that the differenced average weekly foot traffic is an autoregressive process of order 3, which can also be denoted as AR(3).

5.4 Forecasting an autoregressive process

Once the order is determined, we can fit an autoregressive model to forecast our time series. In this case, the model is also termed AR(p), where *p* is still the order of the process.

Here, we will forecast next week's average foot traffic in a retail store using the same dataset we have been working with though this chapter. In order to evaluate our forecasts, we will hold out the last 52 weeks of data for our test set, while the rest will be used for training. That way, we can evaluate the performance of our forecast over a period of one year.

```
df_diff = pd.DataFrame({'foot_traffic_diff': foot_traffic_diff})      #A

train = df_diff[:-52]     #B
test = df_diff[-52:]      #C

print(len(train)) #D
print(len(test))  #E
```

#A Create a DataFrame from the differenced foot traffic data
#B The train set is everything from the beginning except the last 52 data points.

```
#C The test set is only the last 52 data points.
#D Display how many data points are in the train set
#E Display how many data points are in the test set
```

We see that our training set contains 947 data points, while the test set contains 52 data points as expected. Note that the sum of both sets gives 999, which is one less data point from our original series. This is normal, since apply differencing to make the series stationary, and we know that differencing removes the first data point from the series.

Next, we will visualize the testing period for our scenario, in both the original series and differenced series. The plot is shown in figure 5.9.

```
fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=1, sharex=True, figsize=(10, 8)) #A

ax1.plot(df.foot_traffic)
ax1.set_xlabel('Time')
ax1.set_ylabel('Avg. weekly foot traffic')
ax1.axvspan(948, 1000, color='#808080', alpha=0.2)

ax2.plot(df_diff.foot_traffic_diff)
ax2.set_xlabel('Time')
ax2.set_ylabel('Diff. avg. weekly foot traffic')
ax2.axvspan(947, 999, color='#808080', alpha=0.2)

plt.xticks(np.arange(0, 1000, 104), np.arange(2000, 2020, 2))

fig.autofmt_xdate()
plt.tight_layout()
```

#A Specify the figure's size in using the `figsize` parameter. The first number is the height, and the second number is the width in inches.

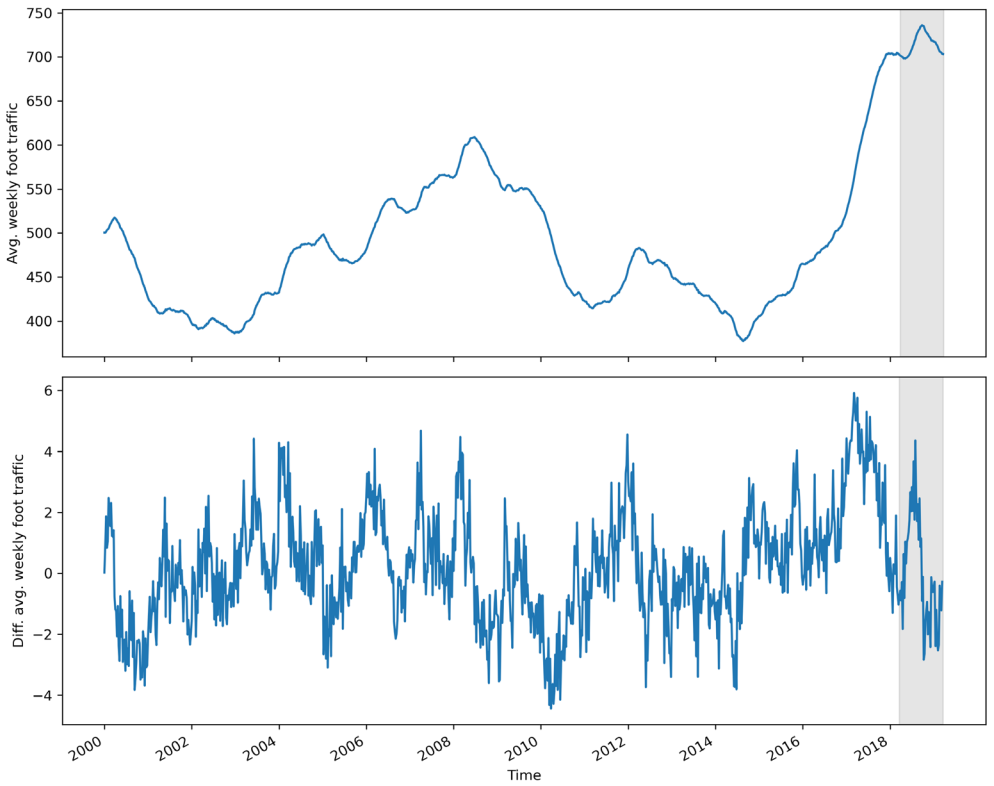


Figure 5.9 Testing period for our forecasts on the original and differences series. Keep in mind that our differenced series has lost its first data point.

Given that our objective is to forecast next week's average foot traffic at the retail store, we will perform rolling forecasts over our test set. Remember that our data was recorded over a weekly period, so predicting the next timestep is forecasting next week's average foot traffic.

We will forecast using three different methods: the historical mean method and the last known value method will act as baselines, and we will use an AR(3) model, since we previously established that we have a stationary third-order autoregressive process. As we did in the previous chapter, we will use the mean squared error (MSE) to evaluate the performance of each forecasting method.

Also, we will reuse the function we defined in the previous chapter to recursively forecast over the testing period. Only this time, we must include a method to use an autoregressive model.

We will use again the SARIMAX function from `statsmodels`, as it encompasses an AR model. As mentioned, SARIMAX is a complex model that allows us to consider seasonal effects, autoregressive processes, non-stationary time series, moving average processes,

and exogenous variables all in one single model. For now, we will disregard all factors except the moving autoregressive portion.

Listing 5.1 A function for rolling forecasts on a horizon

```
def rolling_forecast(df: pd.DataFrame, train_len: int, horizon: int, “[CA]”window: int,
       method: str) -> list:

    total_len = train_len + horizon
    end_idx = train_len

    if method == 'mean':
        pred_mean = []

        for i in range(train_len, total_len, window):
            mean = np.mean(df[:i].values)
            pred_mean.extend(mean for _ in range(window))

        return pred_mean

    elif method == 'last':
        pred_last_value = []

        for i in range(train_len, total_len, window):
            last_value = df[:i].iloc[-1].values[0]
            pred_last_value.extend(last_value for _ in range(window))

        return pred_last_value

    elif method == 'AR':
        pred_AR = []

        for i in range(train_len, total_len, window):
            model = SARIMAX(df[:i], order=(3,0,0))  #A
            res = model.fit(disp=False)
            predictions = res.get_prediction(0, i + window - 1)
            oos_pred = predictions.predicted_mean.iloc[-window:]
            pred_AR.extend(oos_pred)

        return pred_AR
```

#A The order specifies an AR(3) model.

Once our function is defined, we can use it to generate the predictions according to each method. We will assign them to their own column in `pred_df`.

```
pred_df = test.copy()     #A

TRAIN_LEN = len(train)    #B
HORIZON = len(test)       #C
WINDOW = 1        #D

pred_mean = rolling_forecast(df_diff, TRAIN_LEN, HORIZON, WINDOW, 'mean')
pred_last_value = rolling_forecast(df_diff, TRAIN_LEN, HORIZON, WINDOW, “[CA]”'last')
pred_AR = rolling_forecast(df_diff, TRAIN_LEN, HORIZON, WINDOW, 'AR')

pred_df['pred_mean'] = pred_mean   #E
pred_df['pred_last_value'] = pred_last_value
```

```
pred_df['pred_AR'] = pred_AR

pred_df.head()
```

#A `pred_df` **is simply a copy of our test set, so that we have the actual values stored in the same** `DataFrame`
#B Store the length of the training set. Note that constants are usually in capital letters in Python.
#C Store the length of the test set.
#D Since we wish to predict the next timestep, our window is therefore 1.
#E Store the predictions into their respective column in `pred_df`.

We can now visualize our predictions against the observed values in the test set. Note that we are working with the differenced series, so our predictions are also differenced values. The result is shown in figure 5.10.

```
fig, ax = plt.subplots()

ax.plot(df_diff.foot_traffic_diff) #A
ax.plot(pred_df.foot_traffic_diff, 'b-', label='actual')     #B
ax.plot(pred_df.pred_mean, 'g:', label='mean')      #C
ax.plot(pred_df.pred_last_value, 'r-.', label='last')        #D
ax.plot(pred_df.pred_AR, 'k--', label='AR(3)')      #E

ax.legend(loc=2)

ax.set_xlabel('Time')
ax.set_ylabel('Diff. avg. weekly foot traffic')

ax.axvspan(947, 999, color='#808080', alpha=0.2)

ax.set_xlim(920, 999)

plt.xticks([936, 988],[2018, 2019])

fig.autofmt_xdate()
plt.tight_layout()
```

#A Plot part of the training set, so we can see the transition from the training set to the test set.
#B Plot the values from the test set.
#C Plot the predictions from the historical mean method
#D Plot the predictions from the last known value method
#E Plot the predictions from the AR(3) model

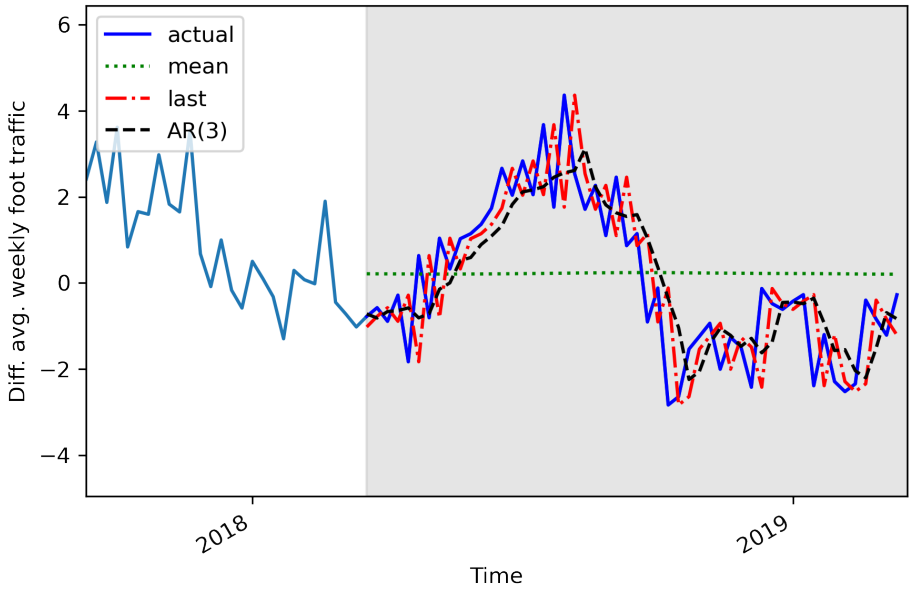


Figure 5.10 Forecasts of the differenced average weekly foot traffic in a retail store.

Looking at figure 5.10, we see that once again, using the historical mean produces a straight line, which shown in the plot as a dotted line. As for the prediction coming from the AR(3) model and the last known value method, the curves are almost confounding with that of the test set, so we will have to measure the MSE in order to assess with method is the most performant. Again, we will use the `mean_squared_error` function from the `sklearn` library.

```
from sklearn.metrics import mean_squared_error

mse_mean = mean_squared_error(pred_df.foot_traffic_diff, pred_df.pred_mean)
mse_last = mean_squared_error(pred_df.foot_traffic_diff, pred_df.pred_last_value)
mse_AR = mean_squared_error(pred_df.foot_traffic_diff, pred_df.pred_AR)

print(mse_mean, mse_last, mse_AR)
```

This prints out a MSE of 3.11 for the historical mean method, 1.45 for the last known value method, and 0.92 for the AR(3) model. Since the MSE for the AR(3) model is lowest of all, we conclude that the AR(3) model is the best performing method to forecast next week's average foot traffic. This is expected, since we established that our stationary process was a third-order autoregressive process. Therefore, it makes sense that modeling using an AR(3) model yield the best predictions over naïve forecasting methods.

Since our forecasts are differenced values, we need to reverse the transformation in order to bring our forecasts back to the original scale of the data otherwise our predictions do not make sense in its business context.

Thus, we will take the cumulative sum of our predictions and add it to the last value of our training set in the original series. This point occurs at index 948, since we are forecasting the last 52 weeks in a dataset containing 1000 points.

```
df['pred_foot_traffic'] = pd.Series()
df['pred_foot_traffic'][948:] = df['foot_traffic'].iloc[948] + pred_df['pred_AR'].cumsum()
         #A
```

#A Assign the undifferenced predictions to the *pred_foot_traffic* column in `df`.

Now, we can plot our undifferenced predictions against the observed values in the test set of the original series in its original scale. The result is shown in figure 5.11.

```
fig, ax = plt.subplots()

ax.plot(df.foot_traffic)
ax.plot(df.foot_traffic, 'b-', label='actual')      #A
ax.plot(df.pred_foot_traffic, 'k--', label='AR(3)') #B

ax.legend(loc=2)

ax.set_xlabel('Time')
ax.set_ylabel('Average weekly foot traffic')

ax.axvspan(948, 1000, color='#808080', alpha=0.2)

ax.set_xlim(920, 1000)
ax.set_ylim(650, 770)

plt.xticks([936, 988],[2018, 2019])

fig.autofmt_xdate()
plt.tight_layout()
```

#A Plot the actual values
#B Plot the undifferenced predictions

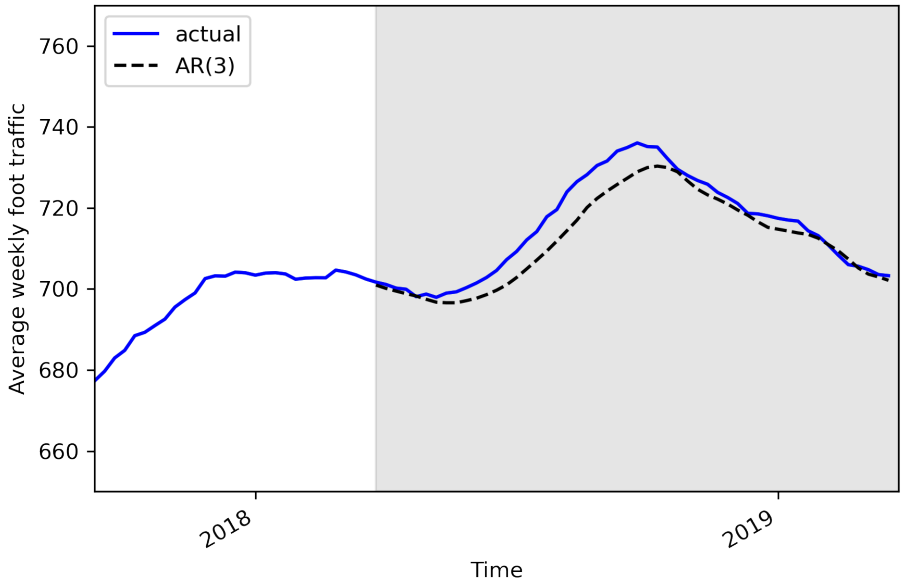


Figure 5.11 Undifferenced forecasts from the AR(3) model.

Looking at figure 5.11, we can that our model (shown as a dashed line) follows the general trend of the observed values in the test set.

Now, we can measure the MSE on the original dataset to get its meaning in a business context. Therefore, we simply measure the MSE again, only using the undifferenced predictions.

```
mse_AR_undiff = mean_squared_error(df.foot_traffic[948:], df.pred_foot_traffic[948:])

print(mse_AR_undiff)
```

This prints out a mean squared error of 20.4. Since the MSE is a squared error, we can take its square root and obtain 4.5. This means that, on average, our predictions are off by 4.5 people on average, either above or below the actual value for the week's average foot traffic.

5.5 Next steps

In this chapter, we covered the autoregressive process and how it can be modeled by an AR(p) model, where *p* is the order and it determines how many lagged values are included in the model. We also saw how plotting the ACF cannot help us determine the order of a stationary AR(p) process. Instead, we must plot the PACF in which the partial autocorrelation coefficients will be significant up until lag *p* only.

However, there might be a situation where neither the ACF or PACF will gives us information. What if both the ACF and PACF plots exhibit a slow decay or a sinusoidal

pattern? In that case, there is no order for the MA(q) or AR(p) process that can be inferred. This means that we are facing a more complex process that is likely a combination of both an AR(p) process and a MA(q) process. This is called an **autoregressive moving average process**, or **ARMA(p,q)**, and it will the subject of the next chapter.

5.6 Exercises

5.6.1 Easy: Simulate an AR(2) process and make forecasts

- Simulate a stationary AR(2) process. Use the `ArmaProcess` function from the `statsmodels` library and simulate the following process:
- $y_t = 0.33y_{t-1} + 0.50y_{t-2}$
- For this exercise, generate 1000 samples.

```
from statsmodels.tsa.arima_process import ArmaProcess
import numpy as np

np.random.seed(42)        #A

ma2 = np.array([1, 0, 0])
ar2 = np.array([1, -0.33, -0.50])

AR2_process = ArmaProcess(ar2, ma2).generate_sample(nsample=1000)
```

#A Set the seed for reproducibility. There is randomness every time you generate samples. By setting the seed, you make sure that the same values are obtained no matter how many times the code is run. Change the seed if you want to experiment with different values.

- Plot your simulated autoregressive process.
- Run the ADF test and check if the process is stationary. If not, apply differencing.
- Plot the ACF. Is it slowly decaying?
- Plot the PACF. Are there significant coefficients after lag 2?
- Separate your simulated series into a training and test set. Take the first 800 timesteps for the training set and assign the rest to the test set.
- Make forecasts over the test set. Use the historical mean method, last value method, and an AR(2) model. Use the `recursive_forecast` function and use a `window` length of 2.
- Plot your forecasts
- Measure the MSE and identify your champion model
- Plot your MSEs in a bar plot

5.6.2 Medium: Simulate an AR(p) process and make forecasts

- Recreate the previous exercise but simulating an AR(p) process of your choice. Experiment with a third or fourth-order autoregressive process. I would recommend generating 10,000 samples.
- When forecasting, experiment different values for the `window` parameter of your `recursive_forecast` function. How does it affect the model's performance? Is there a value that minimizes the MSE?

5.7 Summary

- An autoregressive process states that the present value is linearly dependent on its past values and an error term.
- If the ACF plot of a stationary process shows a slow decay, then we likely have an autoregressive process.
- The partial autocorrelation measures the correlation between two lagged values of a time series when we remove the effect of other autocorrelated lagged values.
- Plotting the PACF of a stationary autoregressive process will show the order *p* of the process. The coefficients will be significant up until lag *p* only.

6

Modeling complex time series

This chapter covers

- Examining the autoregressive moving average model or ARMA(p,q)
- Experimenting with the limitations of the ACF and PACF plots
- Selecting the best model with the Akaike's Information Criterion (AIC)
- Analyzing a time series model using residuals analysis
- Building a general modeling procedure
- Forecasting using the ARMA(p,q) model

In chapter 4, we covered the moving average process, denoted as MA(q), where q is the order. We learned that in a moving average process, the present value is linearly dependent on the mean, the current error term, and past error terms. The order q can be inferred using the ACF plot where autocorrelation coefficients will be significant up until lag q only. In the case where the ACF plot shows a slowly decaying pattern or a sinusoidal pattern, then it is possible that we are in the presence of an autoregressive process instead of a moving average process.

This led us to chapter 5 in which we covered the autoregressive process, denoted as AR(p), where p is the order. In the autoregressive process, the present value is linearly dependent on its own past value. In other words, it is a regression of the variable against itself. We saw that we can infer the order p using the PACF plot, where the partial autocorrelation coefficients will be significant up until lag p only.

We are therefore at a point where we can identify, model, and predict a random walk, a pure moving average process, and a pure autoregressive process using the steps outlined in figure 6.1.

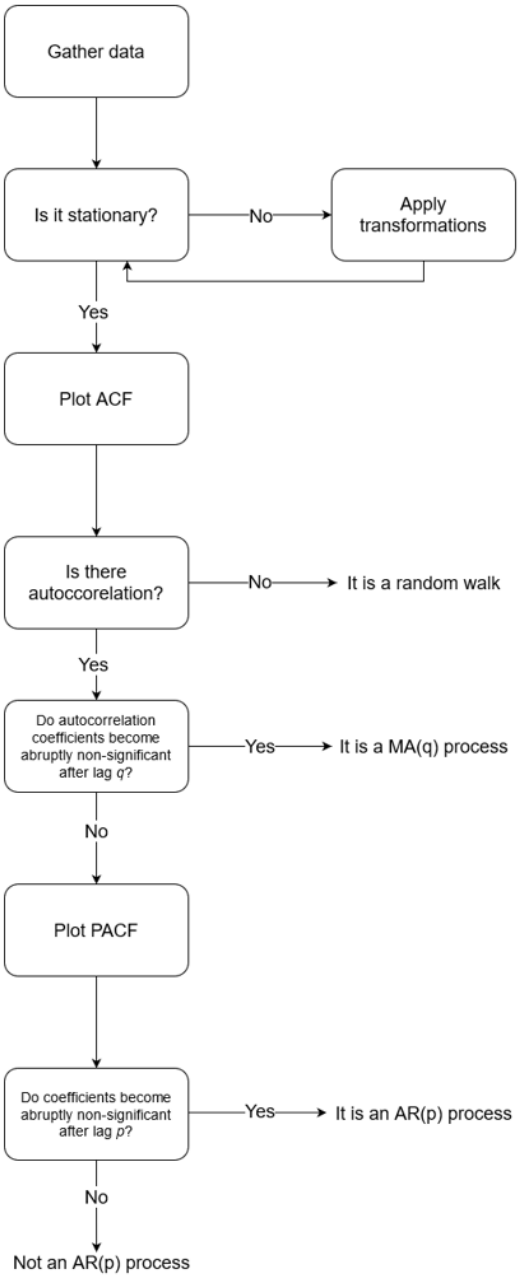


Figure 6.1 Steps to identify and model a random walk, moving average process, and an autoregressive process

The next step is learning how to treat time series where you cannot infer an order from the ACF plot nor from the PACF plot. This means that both figures exhibit a slowly decaying pattern or a sinusoidal pattern. In such case, we are in the presence of an autoregressive moving average process or ARMA. This denotes the combination of both the autoregressive and moving average processes that we covered in the two previous chapters.

In this chapter, we will examine the autoregressive moving average process or ARMA(p,q), where p denotes the order of the autoregressive portion, and q denotes the order of the moving average portion. Furthermore, we will not be able to use the ACF and PACF plots to determine the orders q and p respectively, as both plots will show either a slowly decaying or sinusoidal pattern. Thus, we will define a general modeling procedure that will allow us to model such complex time series. This procedure involves model selection using the Akaike's Information Criterion or **AIC**, which will determine the optimal combination of p and q for our series. Then, we must evaluate the models' validity using residual analysis, by studying the correlogram, Q-Q plot and density plot of the model's residuals in order to assess if they closely resemble white noise. If that is the case, we can move on to forecasting our time series using the ARMA(p,q) model.

This chapter introduced foundational knowledge to forecasting complex time series, as all concepts introduced here will be reused in further chapters when we start modeling non-stationary time series and incorporating seasonality and exogenous variables.

6.1 Forecasting bandwidth usage for data centers

Suppose that you are tasked with predicting bandwidth usage for a large data center. Bandwidth is defined as the maximum rate of data than can be transferred. Its base unit is bits per second (Bps). Forecasting bandwidth usage allows data centers to better manage their computing resources. In the case where a smaller bandwidth usage is expected, then they can shutdown some of their computing resources. This in turns reduces expenses and allows for maintenance. On the other hand, if usage bandwidth usage is expected to increase, then they can dedicate the required resources to sustain the demand and ensure low latency, thus keeping their customers satisfied.

For this situation, we collected 10 000 data points representing the hourly bandwidth usage starting on January 1st, 2019. Here, the bandwidth is measured in megabits per second, or MBps, which is equivalent to 10^6 Bps. We can visualize our time series in figure 6.2.

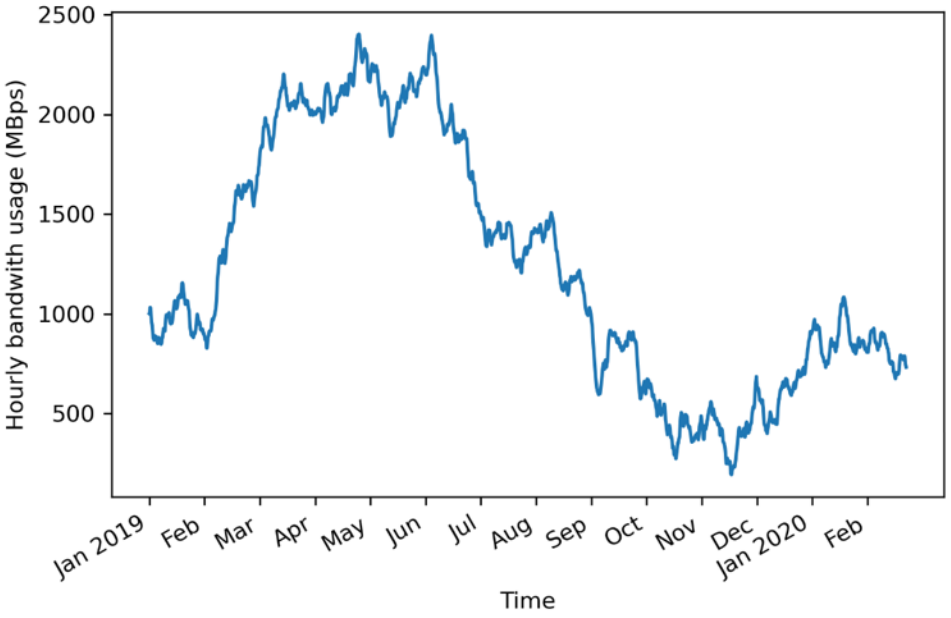


Figure 6.2. Hourly bandwidth usage in a data center since January 1st, 2019. The dataset contains 10,000 points.

Looking at figure 6.2, we can see long-term trends over time, meaning that our series is likely not stationary, and we need to apply a transformation. Also, there seems to be no cyclical behavior, so we can rule out the presence of seasonality in our series.

In order to forecast bandwidth usage, we need to identify the underlying process in our series. Thus, we follow through the steps that we defined in chapter 5. That way, we can verify whether we have a random walk, a moving average process, or an autoregressive process. The steps are shown in figure 6.3.

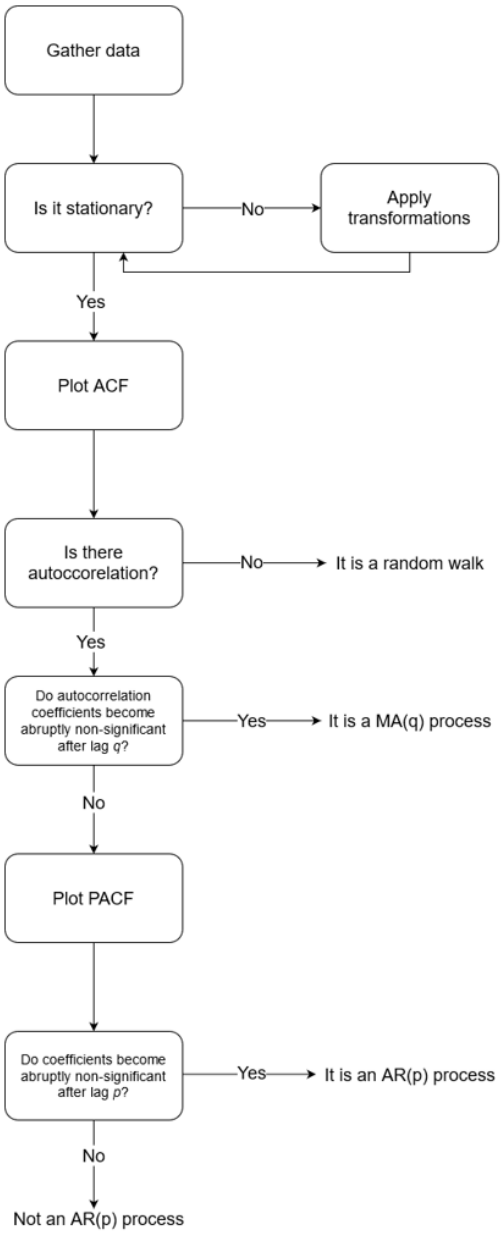


Figure 6.3 Steps to identify a random walk, a moving average process, and an autoregressive process.

Of course, the first step is to collect the data, which is already done in this case. Then, we must determine if our series is stationary or not. The presence of a trend in the plot hints that our series is not stationary. Nevertheless, we will apply the ADF test to test for stationarity and apply a transformation accordingly.

Then, we will plot the ACF function and find out that there are significant autocorrelation coefficients after lag 0. Therefore, it is not a random walk. However, we will observe that coefficients slowly decay. Therefore, they do not become abruptly non-significant after a certain lag, which means that it is not a purely moving average process either.

We then move on to plotting the PACF function. This time, we will notice a sinusoidal pattern, meaning that coefficients do not become abruptly non-significant after a certain lag. This leads us to the conclusion that it is not a purely autoregressive process either.

Therefore, it must be a combination of both the autoregressive process and moving average process, resulting in an autoregressive moving average process that can be modeled with the ARMA(p,q) model, where p is the order of the autoregressive process, and q is the order of the moving average process. However, we cannot use the ACF and PACF plots to respectively find p and q. Thus, we will fit many ARMA(p,q) models with different combination of values for p and q. We will then select a model according to the Akaike's Information Criterion or AIC and assess its viability via the analysis of its residuals. Ideally, the residuals of a model will have similar characteristics to white noise. Then, we will be able to use this model to make forecasts. In this case, we wish to forecast the hourly bandwidth usage for the next seven days.

6.2 Examining the autoregressive moving average process

An autoregressive moving average process is the combination of the autoregressive process and moving average process. It states that the present value is linearly dependent on its own previous values and a constant, just like in an autoregressive process, as well as the mean of the series, the current error term, and past error terms, like in a moving average process.

The autoregressive moving average process is denoted as ARMA(p,q), where p is the order of the autoregressive portion, and q is the order of the moving average portion. Mathematically, the ARMA(p,q) process is expressed as a linear combination of a constant C, the past values of the series y_{t-p}, the mean of the series μ, past error terms ε_{t-q}, and the current error term ε_t, as shown in equation 6.1.

$$y_t=C+\phi_1 y_{t-1}+\phi_2 y_{t-2}+\ldots+\phi_p y_{t-p}+\mu+\epsilon_t+\theta_1\epsilon_{t-1}+\theta_2\epsilon_{t-2}+\ldots+\theta_p\epsilon_{t-q}$$

Equation 6.1

Autoregressive moving average process

An autoregressive moving average process is the combination of the autoregressive process and the moving average process.

It is denoted as ARMA(p,q), where p is the order of the autoregressive process, and q is the order of the moving average process. The general equation of the ARMA(p,q) model is:

$$y_t=C+\phi_1 y_{t-1}+\phi_2 y_{t-2}+...+\phi_p y_{t-p}+\mu+\epsilon_t+\theta_1\epsilon_{t-1}+\theta_2\epsilon_{t-2}+...+\theta_p\epsilon_{t-q}$$

An ARMA(0,q) process is equivalent to a MA(q) process, since the order $p = 0$ cancels the AR(p) portion.

An ARMA(p,0) process is equivalent to an AR(p) process, since the order $q = 0$ cancels the MA(q) portion.

Again, the order p determines the number of past values that affect the present value. Similarly, the order q determines the number of past error terms that affect the present value. In other words, the order p and q respectively dictate the number of parameters for the autoregressive portion and the moving average portion.

Thus, if we have an ARMA(1,1) process, this means that we are combining an autoregressive process of order 1, or AR(1), with a moving a moving average process of order 1, or MA(1). Recall that a first-order autoregressive process is a linear combination of a constant C, the value of the series at the previous timestep $\varphi_1 y_{t-1}$, and white noise ε_t, as shown in equation 6.2.

$$\text{MA}(1):=y_t=C+\phi_1 y_{t-1}+\epsilon_t$$

Equation 6.2

Also recall that a first-order moving average process, or MA(1), is a linear combination of the mean of the series μ, the current error term ε_t, and the error term at the previous timestep $\theta_1\varepsilon_{t-1}$, as shown in equation 6.3.

$$\text{MA}(1):=y_t=\mu+\epsilon_t+\theta_1\epsilon_{t-1}$$

Equation 6.3

We can then combine the AR(1) and MA(1) processes to obtain an ARMA(1,1) process as shown in equation 6.4, which is simply the addition of equation 6.2 with equation 6.3. Notice that the error term is kept as ε_t and not $2\varepsilon_t$, becomes it is a random number, and multiplying it by a factor of 2 will still result in a random number. Therefore, we keep the notation as ε_t.

$$\text{ARMA}(1,1):=y_t=C+\phi_1 y_{t-1}+\mu+\epsilon_t+\theta_1\epsilon_{t-1}$$

Equation 6.4

If we have an ARMA(2,1) process, then we are combining a second-order autoregressive process with a first-order moving average process. We know that we can express an AR(2) process as equation 6.5, while the MA(1) process from equation 6.3 remains the same.

$$\text{AR}(2) := y_t = C + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \epsilon_t$$

Equation 6.5

Then, an ARMA(2,1) process can be expressed as equation 6.6. Notice that it is the combination of the AR(2) process defined in equation 6.5 and the MA(1) process defined in equation 6.3.

$$\text{ARMA}(2,1) := y_t = C + \phi_1 y_{t-1} + \phi_2 y_{t-2} + \mu + \epsilon_t + \theta_1 \epsilon_{t-1}$$

Equation 6.6

In the case where $p = 0$, then we have an ARMA(0,q) process which equivalent to a pure MA(q) process as seen in chapter 4. Similarly, if $q = 0$, then we have an ARMA(p,0) process which is equivalent to a pure AR(p) process as seen in chapter 5.

We can see now how the order p only affects the autoregressive portion of the process by determining the number of past values to include in the equation. Similarly, the order q only affects the moving average portion of the process by determining the number of past error terms to include in the equation of ARMA(p,q). Of course, the higher the orders p and q, the more terms are included, and the more complex our process becomes.

In order to model and forecast an ARMA(p,q) process, we need to find the orders p and q. That way, we can use an ARMA(p,q) model to fit on the available data and produce forecasts.

6.3 Identifying a stationary ARMA process

Now that we understand the definition the autoregressive moving average process and how the orders p and q affect the model's equation, we need to determine a set of steps that will allow us to identify such underlying process in a given time series.

To do so, we extend the steps that we defined in chapter 5 to include the final possibility that we have an ARMA(p,q) process, as shown in figure 6.4.

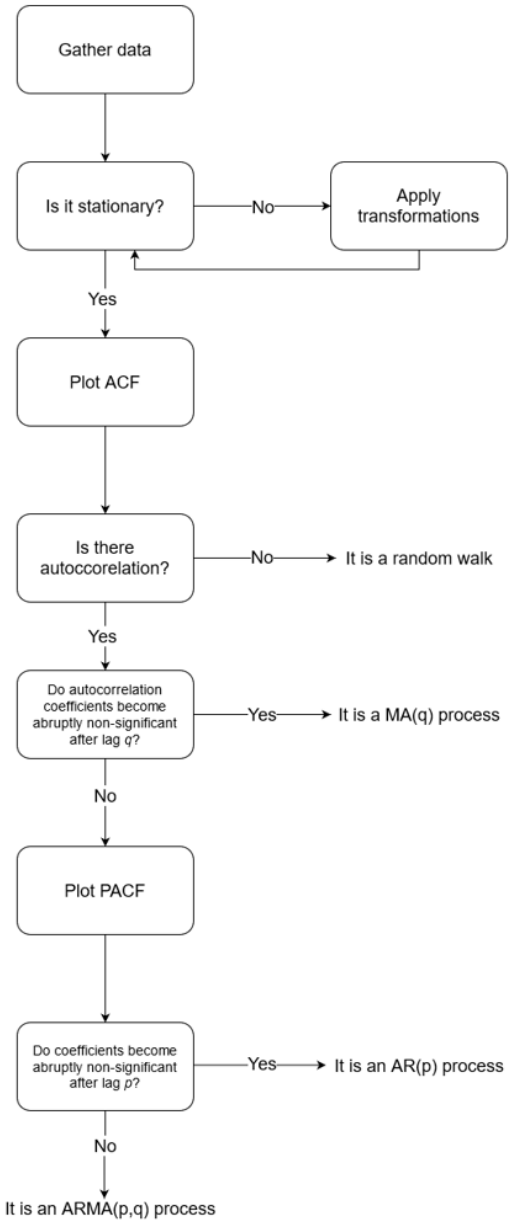


Figure 6.4 Steps to identify a random walk, a moving average process MA(q), an autoregressive process AR(p), and an autoregressive moving average process ARMA(p,q).

Looking at figure 6.4, we notice that if both the ACF and PACF plots do not show a clear cutoff between significant and non-significant coefficients, then we have an ARMA(p,q) process. To verify that, let's simulate our own ARMA process.

Here, we simulate an ARMA(1,1) process. This is equivalent to combining a MA(1) process with an AR(1) process. Specifically, we will simulate the ARMA(1,1) process defined in equation 6.7.

$$y_t = 0.33y_{t-1} + 0.9\epsilon_{t-1} + \epsilon_t$$

Equation 6.7

Notice that in equation 6.7, the constant C and mean μ are both equal to 0. The coefficients 0.33 and 0.9 are subjective choices for the purpose of the simulation.

The objective of this simulation is to demonstrate that we cannot use the ACF plot to identify the order q of an ARMA(p,q) process, which in this case is 1, nor can we use the PACF plot to identify the order p of an ARMA(p,q) process, which in this case is 1 also.

We use the `ArmaProcess` function from the `statsmodels` library to simulate our ARMA(1,1) process. As done in previous chapters, we define the array of coefficients for the AR(1) process, as well as for the MA(1) process. From equation 6.7, we our AR(1) process will have a coefficient of 0.33. However, keep in mind that the function expects to have the coefficient of the autoregressive process with its opposite sign, as this is how it is implemented in the `statsmodels` library. Therefore, we input it as -0.33. For the moving average portion, equation 6.7 specifies that the coefficient is 0.9. Also recall that when defining your arrays of coefficients, the first coefficient is always equal to 1, as specified by the library, which represents the coefficient at lag 0. Once our coefficients are defined, we generate 1000 data points.

```
from statsmodels.tsa.arima_process import ArmaProcess
import numpy as np

np.random.seed(42)

ar1 = np.array([1, -0.33])    #A
ma1 = np.array([1, 0.9])    #B

ARMA_1_1 = ArmaProcess(ar1, ma1).generate_sample(nsample=1000)    #C
```

#A Define the coefficients for the AR(1) portion. Remember that the first coefficient is always 1, as specified by the documentation. Then, we must write the coefficient of the AR portion with the opposite sign of what is defined in equation 6.7.

#B Define the coefficients for the MA(1) portion. The first coefficient is 1 for the 0th lag, as specified by the documentation.

#C Generate 1000 samples

With our simulated data ready, we can move on to the next step and verify if our process is stationary or not. We do by running the Augmented Dickey-Fuller test or ADF test. We print out the ADF statistic as well as the p-value. If the ADF statistic is a large negative number, and if we have a p-value smaller than 0.05, then we can reject the null hypothesis and conclude that we have a stationary process.

```
from statsmodels.tsa.stattools import adfuller

ADF_result = adfuller(ARMA_1_1)    #A

print(f'ADF Statistic: {ADF_result[0]}')
print(f'p-value: {ADF_result[1]}')
```

#A Run the ADF test on the simulated ARMA(1,1) data

This returns an ADF statistic of -6.43 and a p-value of $1.7x10^{-8}$. Since we have a large negative ADF statistic and a p-value that much smaller than 0.05, we can conclude that our simulated ARMA(1,1) process is stationary.

Following the steps outlined in figure 6.4, we plot the ACF and verify if we can infer the order of the moving average portion of our simulated ARMA(1,1) process. Again, we use the `plot_acf` function from `statsmodels` to generate figure 6.5.

```
from statsmodels.graphics.tsaplots import plot_acf

plot_acf(ARMA_1_1, lags=20);

plt.tight_layout()
```

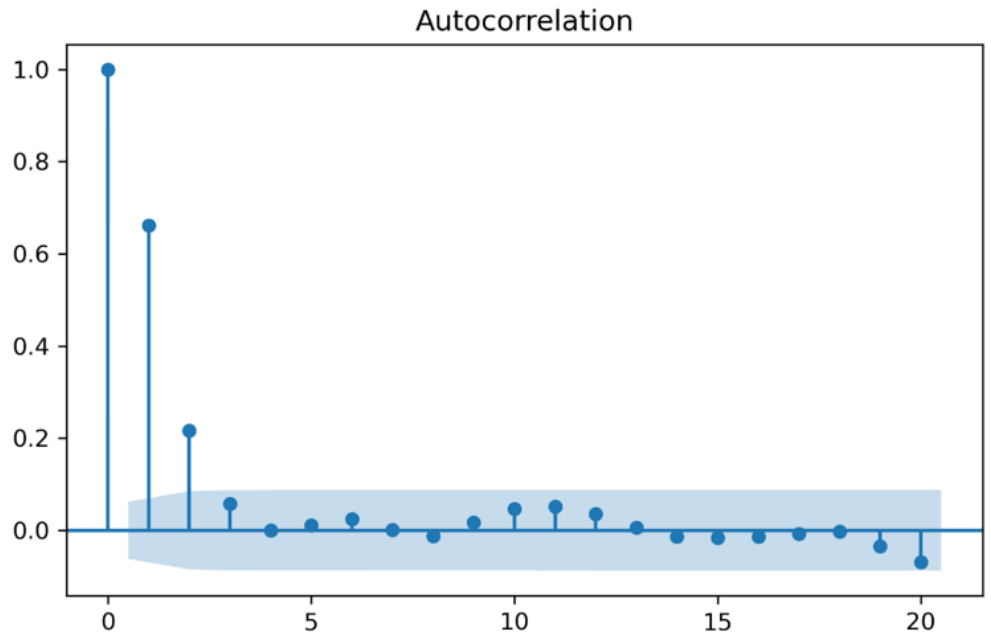


Figure 6.5 ACF plot of our simulated ARMA(1,1) process. Notice the sinusoidal pattern on the plot, meaning that an AR(p) process is in play. Also, the last significant coefficient is at lag 2, meaning that we cannot infer the order q of an ARMA(p,q) process from the ACF plot. We see that last significant coefficient is at lag 2, which suggests that $q = 2$. However, we know that we simulated an ARMA(1,1) process, so q must be equal to

1! Therefore, the ACF plot cannot be used to infer the order q of an ARMA(p,q) process.

Looking at figure 6.5, we notice a sinusoidal pattern in the plot, which indicates the presence of an autoregressive process. This is expected, since we simulated an ARMA(1,1) process and we know the existence of the autoregressive portion. Furthermore, we notice that the last significant coefficient is at lag 2. However, we know that our simulated data has a MA(1) process, and we would expect to have significant coefficients up to lag 1 only.

We can thus conclude that the ACF plot does not reveal any useful information about the order q of our ARMA(1,1) process, where $q = 1$. Therefore, we move to the next step outlined in figure 6.4 and plot the PACF.

In chapter 5, we learned that the PACF can be used to find the order of a stationary AR(p) process. Now, we will verify if we can find the order p of our simulated ARMA(1,1) process, where $p = 1$. We use the `plot_pacf` function to generate figure 6.6.

```
from statsmodels.graphics.tsaplots import plot_pacf

plot_pacf(ARMA_1_1, lags=20);

plt.tight_layout()
```

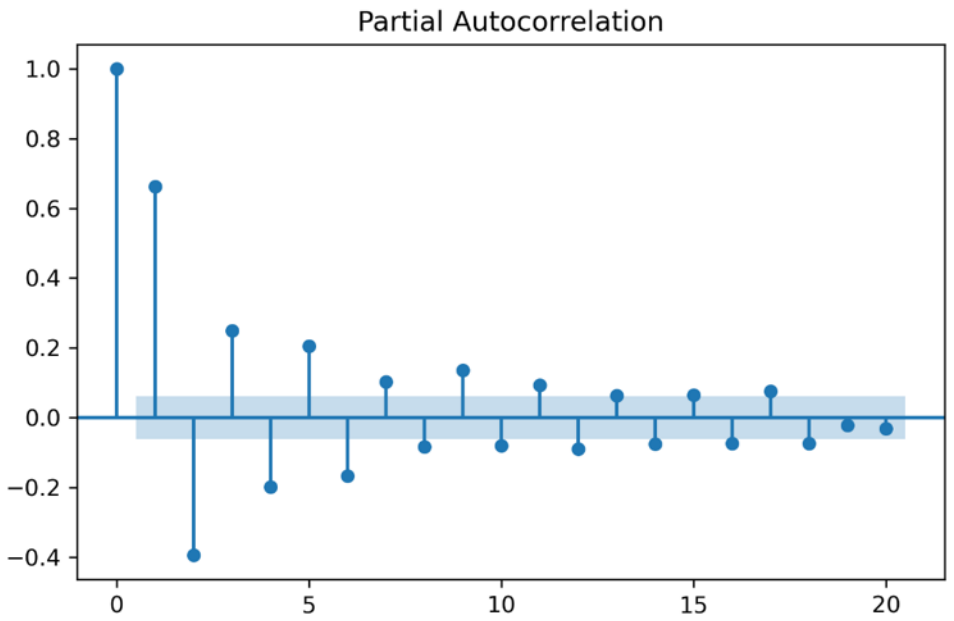


Figure 6.6 PACF plot of our simulated ARMA(1,1) process. Again, we have a sinusoidal pattern with no clear cutoff between significant and non-significant coefficients. From this plot, we cannot infer that $p = 1$ in our simulated ARMA(1,1) process, meaning that we cannot determine the order p of an ARMA(p,q) process using the PACF plot.

From figure 6.6, we can see a clear sinusoidal pattern, meaning that we cannot infer a value for the order p. Whereas we know that we simulated an ARMA(1,1) process, we cannot determine that value from the PACF plot in figure 6.6, since we have significant coefficients past lag 1. Therefore, the PACF cannot be used to find the order p of an ARMA(p,q) process.

Nevertheless, according to figure 6.4, since there is not clear cutoff between significant and non-significant coefficients in both the ACF and PACF plots, we can conclude that we have an ARMA(p,q) process, which is indeed the case.

Identifying a stationary ARMA(p,q) process

If your process is stationary and both the ACF and PACF plots show a decaying pattern or sinusoidal pattern, then it is a stationary ARMA(p,q) process.

We have simulated an ARMA(1,1) process and followed through the steps of figure 6.4 to reach the right conclusion that we have a stationary ARMA(p,q) process, since both the ACF and PACF plots showed sinusoidal patterns and did not reveal the orders q and p respectively.

Now, we know that determining the order of our process is key in modeling and forecasting, since the order will dictate how many parameters must be included in our model. Since the ACF and PACF plots are not useful in the case of an ARMA(p,q) process, we must thus devise a general modeling procedure that will allow us to find the appropriate combination of (p,q) for our model.

6.4 Devising a general modeling procedure

In the previous section, we covered the steps to identify a stationary ARMA(p,q) process. We saw that if both the ACF and PACF plots display a sinusoidal or decaying pattern, then our time series can be modeled by an ARMA(p,q) process. However, neither plot was useful to determine the orders p and q. With our simulated ARMA(1,1) process, we noticed that coefficients were significant after lag 1 in both plots.

Therefore, we must devise a procedure that allows us to find the orders p and q. This procedure has the advantage that it can also be applied in situations where our time series is non-stationary and has seasonal effects. Furthermore, it will also be suitable for cases where p or q are equal to 0, meaning that we can move away from plotting the ACF and PACF, and rely entirely on a model selection criterion and residuals analysis. The steps are shown in figure 6.7.

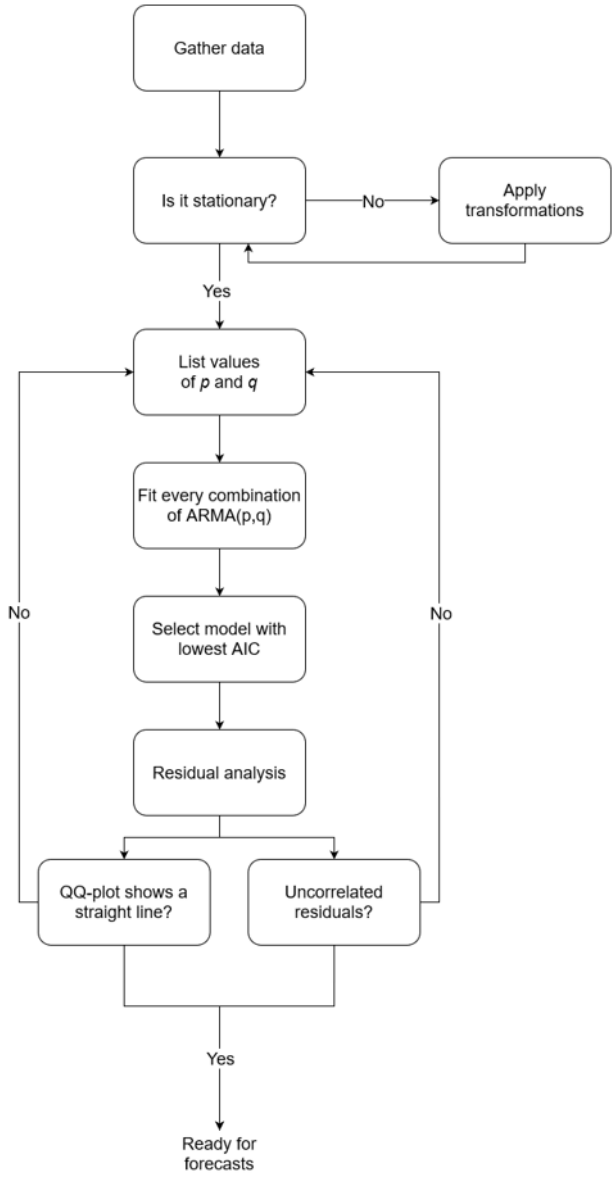


Figure 6.7 General modeling procedure for an ARMA(p,q) process. The first step is to gather the data, test for stationarity, and apply transformations accordingly. Then, we define a list of possible values for p and q. We then fit every combination of ARMA(p,q) on our data and select the model with the lowest AIC. Then, we perform the residual analysis by looking at the Q-Q plot and the residuals' correlogram. If they approach that of white noise, then the model can be used for forecasts. Otherwise, we must try different values for p and q.

Looking at figure 6.7, we see that this new modeling procedure completely removes the plotting of the ACF and PACF. This allows us to select a model based entirely on statistical tests and numerical criteria, instead of relying on the qualitative analysis of the ACF and PACF plots.

The first few steps remain unchanged from what we gradually built up until chapter 5, as we must still gather the data, test for stationarity, and apply transformations accordingly. Then, we will list different possible values of p and q. Note that they only take positive integers. With a list of possible values, can fit every unique combination of ARMA(p,q) on our data. Then, we can compute the **Akaike's Information Criterion**, or **AIC** which is discussed at length in section 6.4.1 and 6.4.2. This quantifies the quality of each model in relation to each other. The model with lowest AIC is then selected. From there, we can analyze the model's residuals, which is the difference between the actual and predicted values of the model. Ideally, the residuals look like white noise, as it would mean that any difference between the predicted values and actual values are due to randomness. Therefore, the residuals must be uncorrelated and independently distributed. We can assess those properties by studying the **Q-Q plot** and running the **Ljung-Box test** which we explore in section 6.4.3. If the analysis leads us to conclude that the residuals are completely random, then we have a model ready for forecasting. Otherwise, we must try a different set of values for p and q, and start the process over.

Now, there are a lot of new concepts and techniques that will be introduced as we work through our new general modeling procedure. We will dive into each step in detail in future sections and work with our simulated ARMA(1,1) process. Then, we will apply the same procedure to model bandwidth usage.

6.4.1 Understanding the Akaike's Information Criterion (AIC)

Before covering the steps outlined in figure 6.7, we need to determine how we will choose the best model from all the models that we will fit. Here, we will use the Akaike's Information Criterion or AIC to select the optimal model.

The AIC estimates the quality of a model relative to other models. This is why it is used for model selection. Given that there will be some information lost when a model is fit to the data, the AIC quantifies the relative amount of information lost by the model. Therefore, the less information lost, the better the model. Consequently, a lower AIC value translates into a better model.

Specifically, the AIC is a function of the number of estimated parameters k and the maximum value of the likelihood function for the model $\hat{L}$, as shown in equation 6.8.

$$\text{AIC}=2\text{k-}2\ln(\hat{L})$$

Equation 6.8

Akaike's Information Criterion (AIC)

The Akaike's Information Criterion, or AIC, is a measure of the quality of a model in relation to other models. It is used for model selection.

The AIC is a function of the number of parameters k in a model and the maximum value of the likelihood function $\hat{L}$:

$$AIC = 2k - 2\ln(\hat{L})$$

The lower the value of the AIC, the better the model. Selecting according to the AIC allows us to keep a balance between the complexity of a model and its goodness of fit to the data.

The number of estimated parameters k is directly related to the order (p,q) of an ARMA(p,q) model. If we fit an ARMA(2,2) model, then we have $2 + 2 = 4$ parameters to estimate. If we fit an ARMA(3,4) model, then we have $3 + 4 = 7$ parameters to estimate. We can see how fitting a more complex model can penalize the AIC score: as the order (p,q) increases, the number of parameters k increases, and so the AIC increases.

The likelihood function measures the goodness of fit of a model. It can be viewed as the opposite of the distribution function. Given a model with fixed parameters, the distribution function will measure the probability of observing a data point. The likelihood function flips the logic. Given a set of observed data, it will estimate how likely different model parameters generate the observed data.

For example, consider the situation where we roll a 6-sided die. The distribution function will tell us that there is a 1/6 probability that we observe either one of these values: *[1,2,3,4,5,6]*. Now, we flip this logic to explain the likelihood function. Suppose that you roll a die 10 times and you obtain the following values: *[1,5,3,4,6,2,4,3,2,1]*. Then, the likelihood function will determine how likely it is that the die has 6 sides. Applying this logic to the context of AIC, we can think of the likelihood function as an answer to the question *How likely is my observed data coming from an ARMA(1,1) model?* If it is very likely, meaning that $\hat{L}$ is large, then the ARMA(1,1) model fits the data well.

Therefore, if a model fits the data really well, then the maximum value of the likelihood will be high. Since the AIC subtracts the natural logarithm of the maximum value of the likelihood, represented by $\hat{L}$ in equation 6.8, then a large value of $\hat{L}$ will lower the AIC.

We can see how the AIC keeps a balance between underfitting and overfitting. Remember that the lower the AIC, the better the model relative to other models. Therefore, an overfitting model would have a very good fit, meaning that $\hat{L}$ is large and AIC decreases. However, the number of parameters k would be large as well, which penalizes the AIC.

An underfitting model would have a small number of parameters, so k would be small. However, the maximum value of the likelihood function would also be small due to the poor fit, meaning again that the AIC is penalized.

Thus, the AIC allows us to find a balance between the number of parameters in a model and a good fit to the training data.

Finally, we must keep in mind that the AIC quantifies the quality of a model in relation to other models only. It is therefore a relative measure of quality. In the event that we fit only poor models to our data, the AIC will simply help us determine the best from the group of models.

Now, let's use the AIC to help us select an appropriate model for a simulated ARMA(1,1) process.

6.4.2 Selecting a model using the AIC

We know cover the steps of the general modeling procedure outlined in figure 6.7 using our simulated ARMA(1,1) process.

In the previous section, we already tested for stationarity and concluded that our simulated process is already stationary. Therefore, we can move on to defining a list of possible values for p and q. While we know the values of both orders from the simulation, let's consider the following steps as a demonstration that the general modeling procedure works.

Here, we will allow the values of p and q to vary between 0 and 3. Note that this range is arbitrary, and you may try a larger range of values if you wish. The possible values are then 0, 1, 2 and 3. Then, we will create a list of all possible combinations of (p,q), using the `product` function from `itertools`. Since there four possible values for p and q, this will generate a list of 16 unique combinations of (p,q).

```
from itertools import product

ps = range(0, 4, 1)    #A
qs = range(0, 4, 1)    #B

order_list = list(product(ps, qs))    #C
```

#A Create a list of possible values for p starting from 0 inclusively to 4 exclusively, with steps of 1.
#B Create a list of possible values for q starting from 0 inclusively to 4 exclusively, with steps of 1.
#C Generate a list containing all unique combination of (p,q).

With our list of possible values created, we must now fit all unique 16 ARMA(p,q) model on our simulated data. To do so, we define the function `optimize_ARMA`, that takes the data and the list of unique (p,q) combinations as input. Inside the function, we initialize an empty list to store the (p,q) combination and its corresponding AIC. Then, we iterate over each (p,q) combination and fit an ARMA(p,q) model on our data. We compute the AIC and store the result. Then, we create a `DataFrame` and sort it by AIC value in ascending order, since the lower the AIC, the better the model. Our function finally outputs the ordered `DataFrame` so we can select the appropriate model. The `optimize_ARMA` function is shown in listing 6.1.

Listing 6.1 Function to fit all unique ARMA(p,q) models

```
from typing import Union
from tqdm import tqdm_notebook
from statsmodels.tsa.statespace.sarimax import SARIMAX

def optimize_ARMA(endog: Union[pd.Series, list], order_list: list) -> "[CA]"pd.DataFrame:
      #A

    results = []    #B

    for order in tqdm_notebook(order_list):    #C
        try:
            model = SARIMAX(endog, order=(order[0], 0, order[1]),
       "[CA]"simple_differencing=False).fit(disp=False)    #D
        except:
            continue

        aic = model.aic    #E
        results.append([order, aic])    #F

    result_df = pd.DataFrame(results)    #G
    result_df.columns = ['(p,q)', 'AIC']    #H

    #Sort in ascending order, lower AIC is better
    result_df = result_df.sort_values(by='AIC',
       "[CA]"ascending=True).reset_index(drop=True)    #I

    return result_df
```

#A The function takes as inputs the time series data, and the list of unique (p,q) combinations.
#B Initialize an empty list to store the order (p,q) and its corresponding AIC as a tuple.
#C Iterate over each unique (p,q) combination. The use of `tqdm_notebook` will display a progress bar.
#D Fit an ARMA(p,q) model using the `SARIMAX` function. We specify `simple_differencing=False` to prevent differencing. Recall that differencing is the result of $y_t - y_{t-1}$ We also specify `disp=False` to avoid printing convergence messages to the console.
#E Calculate the model's AIC
#F Append the (p,q) combination and AIC as a tuple to the `results` list.
#G Store the (p,q) combination and AIC in a `DataFrame`.
#H Label the columns of your `DataFrame`.
#I Sort the `DataFrame` in ascending order of AIC values. The lower the AIC, the better the model.

With our function defined, we can now use it and fit the different ARMA(p,q) models. The output is shown in figure 6.8.

```
result_df = optimize_ARMA(ARMA_1_1, order_list)    #A
result_df    #B
```

#A Fit the different ARMA(p,q) models on the simulated ARMA(1,1) data.
#B Display the resulting `DataFrame`.

	(p,q)	AIC
0	(1, 1)	2801.407785
1	(2, 1)	2802.906070
2	(1, 2)	2802.967762
3	(0, 3)	2803.666793
4	(1, 3)	2804.524027
5	(3, 1)	2804.588567
6	(2, 2)	2804.822282
7	(3, 3)	2805.947168
8	(2, 3)	2806.175380
9	(3, 2)	2806.894930
10	(0, 2)	2812.840730
11	(0, 1)	2891.869245
12	(3, 0)	2981.643911
13	(2, 0)	3042.627787
14	(1, 0)	3207.291261
15	(0, 0)	3780.418416

Figure 6.8 Resulting `DataFrame` from fitting all ARMA(p,q) models on the simulated ARMA(1,1) process. We can see that the model with the lowest AIC corresponds to an ARMA(1,1) model, meaning that we successfully identified the order of our simulated data.

Looking at figure 6.8, we notice that the model with the lowest AIC corresponds to an ARMA(1,1) model, which is exactly the process that we simulated.

As mentioned in the previous section, the AIC is measure of relative quality. Here, we can say that an ARMA(1,1) is the best model relative to all other models that we fit to our data. Now, we need an absolute measure of the model's quality. This brings us to the next step of our modeling procedure, which is residuals analysis.

6.4.3 Understanding residuals analysis

Up to this point, we fit different ARMA(p,q) models to our simulated ARMA(1,1) process. Using the AIC as a model selection criterion, we obtained that an ARMA(1,1) model is best model relative to all others that were fit. Now, we must measure its absolute quality by performing an analysis on the model's residuals.

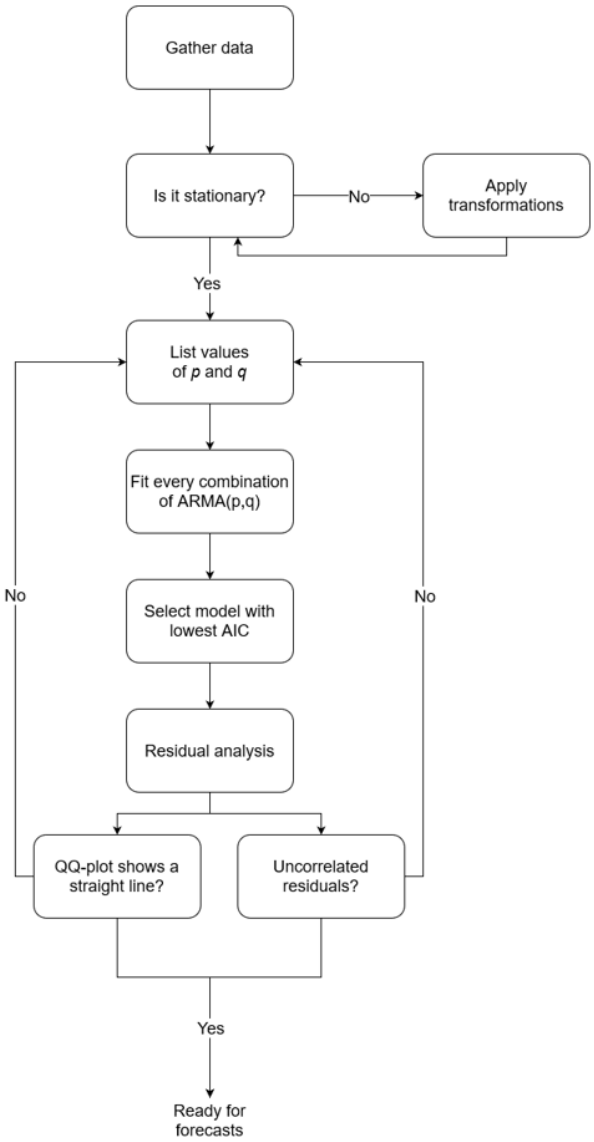


Figure 6.9 General modeling procedure for an ARMA(p,q) process.

This brings us to the last steps before forecasting, which is residuals analysis and answering two questions: does the Q-Q plot show a straight line and are the residuals uncorrelated. If the answer to both questions is yes, then we have a model ready to make forecasts. Otherwise, we must try different combinations of (p,q) and restart the process.

The residuals of a model is simply the difference between the predicted values and the actual values. Consider our simulated ARMA(1,1) process expressed in equation 6.9.

$$y_t=0.33y_{t-1}+0.9\epsilon_{t-1}+\epsilon_t$$

Equation 6.9

Now, suppose that we fit an ARMA(1,1) model to our process, and we estimate the model's coefficients perfectly, such that the model is expressed as equation 6.10.

$$y_t=0.33y_{t-1}+0.9\epsilon_{t-1}$$

Equation 6.10

Then, the residuals will be the difference between the values coming from our model and the observed values from our simulated process. In other words, it is the difference between equation 6.9 and equation 6.10. The result is shown in equation 6.11.

$$\text{residuals}=0.33y_{t-1}+0.9\epsilon_{t-1}+\epsilon_t-(0.33y_{t-1}+0.9\epsilon_{t-1})$$
$$\text{residuals}=\epsilon_t$$

Equation 6.11

As you can see in equation 6.11, in a perfect situation, the residuals of a model are white noise. This is indicative that the model has captured all predictive information, and there is only a random fluctuation left that cannot be modeled. Thus, the residuals must be uncorrelated and have a normal distribution in order for us to conclude that we have a good model to make forecasts.

There are two aspects to residuals analysis: a qualitative analysis and a quantitative analysis. The qualitative analysis focuses on studying the Q-Q plot, while the quantitative analysis determines if our residuals are uncorrelated.

QUALITATIVE ANALYSIS: STUDYING THE Q-Q PLOT

The first step in residuals analysis is the study of the quantile-quantile plot or Q-Q plot. The Q-Q plot is a graphical tool to verify our hypothesis that the model's residuals are normally distributed.

The Q-Q plot is constructed by plotting the quantiles of our residuals on the y-axis, against the quantiles of a theoretical distribution, in this case the normal distribution, on the x-axis. This results in a scatter plot. Remember that we compare the distribution to a normal distribution, because we want the residuals to be similar to white noise, which is normally distributed.

If both distributions are similar, meaning that the distribution of the residuals is close to a normal distribution, then the Q-Q plot will display a straight line that approximately lies on y

$= x$. We can see an example of a Q-Q plot where the residuals are normally distributed in figure 6.10.

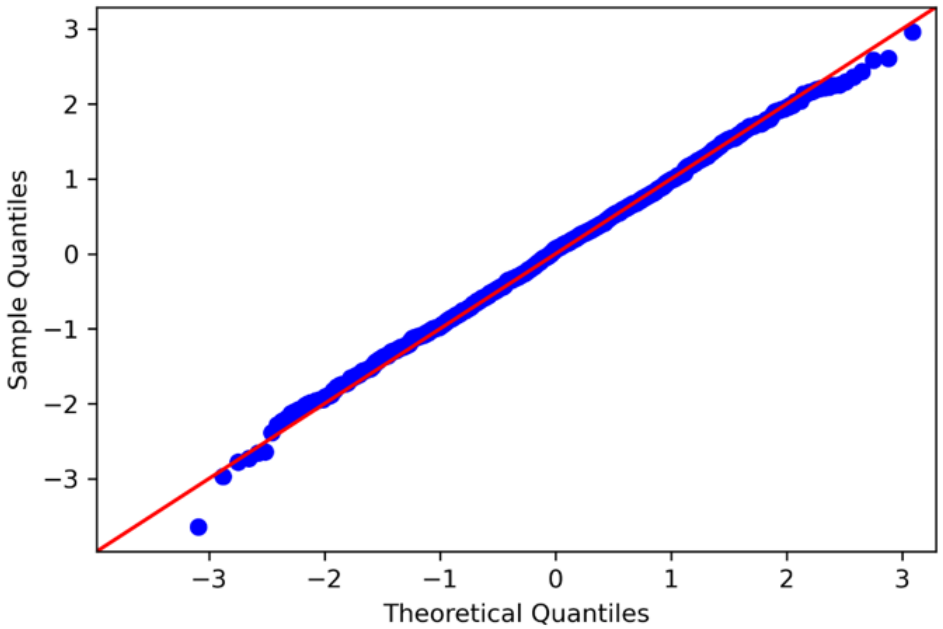


Figure 6.10 Q-Q plot of a randomly distributed residuals. On the y-axis, we have the quantiles coming from the residuals. On the y-axis, we have the quantiles coming from a theoretical normal distribution. We can see a straight line approximately lying on $y = x$. This is an indication that our residuals are very close to a normal distribution.

Looking at figure 6.10, we can see a thick straight line that is roughly lying on $y = x$. Therefore, we can conclude that the residuals are approximately normally distributed. This in turn means that our model is a good fit for our data.

On the other hand, the Q-Q plot of residuals that are not close to a normal distribution will generate a curve that departs from $y = x$ as shown in figure 6.11.

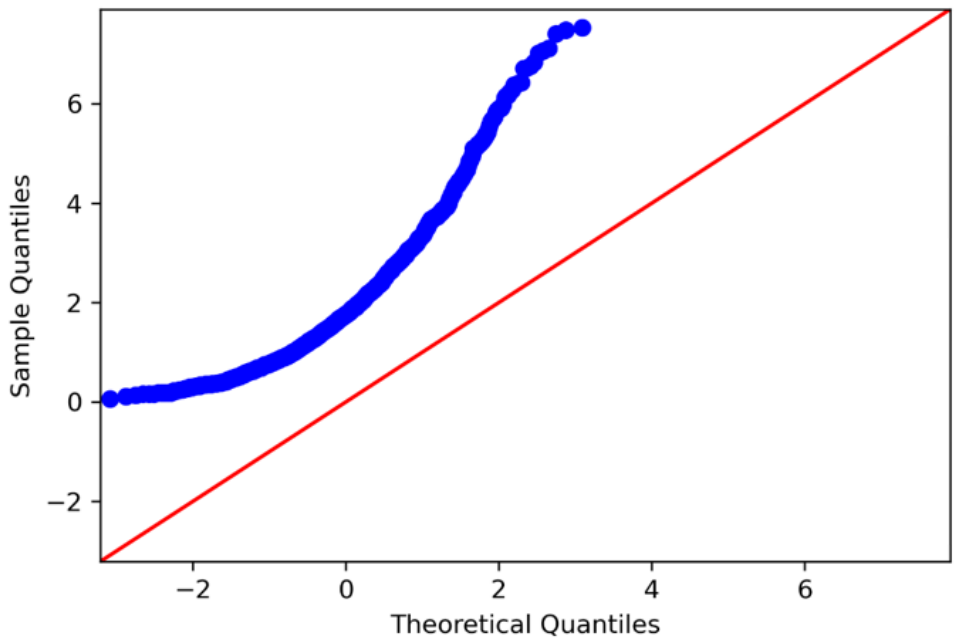


Figure 6.11 Q-Q plot of residuals that are not close to a normal distribution. We can clearly see that the thick line is curved, and it is not lying on $y = x$. Therefore, the distribution of the residuals is very different from a normal distribution.

Looking at figure 6.11, we can see that the thick line is not straight and not lying on $y = x$. Therefore, we can conclude that the distribution of our residuals does not resemble a normal distribution. This is a sign that our model is not a good fit for our data. Therefore, we must try a different range of values for p and q, fit the models, select the one with the lowest AIC and perform residual analysis on the new model.

Quantile-quantile plot (Q-Q plot)

A Q-Q plot is plot of the quantiles of two distributions against each other. In time series forecasting, we plot the distribution of our residuals on the y-axis, against the theoretical normal distribution on the x-axis.

This graphical tool allows to us to assess the goodness of fit of our model. If the distribution of our residuals is similar to a normal distribution, then we will see a straight line lying on $y = x$. This means that our model is a good fit, because the residuals are similar to white noise.

On the contrary, if the distribution of our residuals is different from a normal distribution, then we will see a curved line. We then conclude that our model is not a good fit, since the residuals' distribution is not close to a normal distribution, and therefore the residuals are not similar to white noise.

We can see how the Q-Q plot is a graphical tool to help us assess the goodness of fit of our model. We know that if a model is a good fit to our data, then the residuals be similar to white noise, and therefore will have similar properties. This means that they should be normally distributed. Hence, if the Q-Q plot displays a straight line, then we have a good model. Otherwise, our model must be discarded, and we must try to fit a better model.

While the Q-Q plot is a fast method to assess the quality of our model, this analysis remains subjective. Thus, we will further support our residuals analysis with a quantitative method by applying the Ljung-Box test.

QUANTITATIVE ANALYSIS: APPLYING THE LJUNG-BOX TEST

Once we analyzed the Q-Q plot and determined that our residuals are approximately normally distributed, we can then apply the Ljung-Box test to demonstrate that the residuals are uncorrelated.

Remember that a good model has residuals that are similar to white noise. Therefore, the residuals should be normally distributed and uncorrelated.

The Ljung-Box test is a statistical test that tests if the autocorrelation of a group of data is significantly different from 0. In our case, we apply the Ljung-Box test to the model's residuals to assess if they are correlated or not. The null hypothesis states that the data is independently distributed, meaning that there is no autocorrelation.

Ljung-Box test

The Ljung-Box test is a statistical test that tests if the autocorrelation of a group of data is significantly different from 0.

The null hypothesis states that the data is independently distributed, meaning that there is no autocorrelation.

In time series forecasting, we apply the Ljung-Box test on the model's residuals to test if they are similar to white noise. If the p-value is larger than 0.05, then we cannot reject the null hypothesis, meaning that the residuals are independently distributed. Therefore, there is no autocorrelation, the residuals are similar to white noise, and the model can be used for forecasting.

Otherwise, if the p-value if less than 0.05, then we reject the null hypothesis, meaning that our residuals are not independently distributed and are correlated. The model cannot be used for forecasting.

The test will return the Ljung-Box statistic and a p-value. If the p-value is less than 0.05, then we reject the null hypothesis, meaning that the residuals are not independently distributed, which in turn means that there is autocorrelation. In such situation, the residuals do not approximate the properties of white, and the model must be discarded.

If the p-value is larger than 0.05, then we cannot reject the null hypothesis, meaning that our residuals are independently distributed. Thus, there is not autocorrelation, and the residuals are similar to white noise. This means that we can move on with our model and make forecasts.

Now that we understand the concepts of residuals analysis, let's apply these techniques to our simulated ARMA(1,1) process.

6.4.4 Performing residuals analysis

We will now resume the modeling procedure of our simulated ARMA(1,1) process. Looking at figure 6.12, we have successfully selected a model with the lowest AIC, which was expectedly an ARMA(1,1) model. Now, we will perform residuals analysis to assess if our model is a good fit to the data.

While we know that our ARMA(1,1) model must be good, since we simulated an ARMA(1,1) process, we will use this section as a demonstration that our modeling procedure works. We are likely not going to be modeling and forecasting simulated data in a business context, so it is important to cover the entire modeling procedure on a known process first, to convince ourselves that it works, before applying it on real-life data.

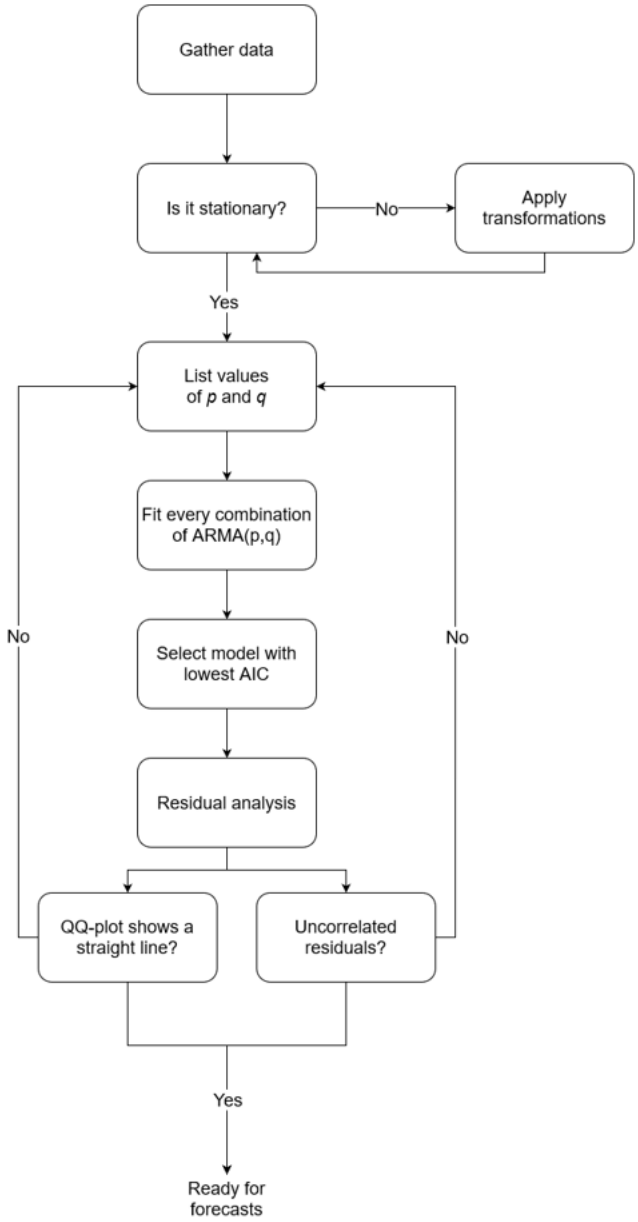


Figure 6.12 General modeling procedure for an ARMA(p,q) process.

To perform residuals analysis, we need to fit our model and store the residuals in a variable for easy access. Using `statsmodels`, we first define an ARMA(1,1) model before fitting it to our simulated data. Then, we can access the residuals with the `resid` property.

```
model = SARIMAX(ARMA_1_1, order=(1,0,1), simple_differencing=False)
model_fit = model.fit(disp=False)
residuals = model_fit.resid    #A
```

#A Store the model's residuals

The next step is to plot the Q-Q plot. We use the `qqplot` function from `statsmodels` to display the quantile-quantile plot of our residuals against a normal distribution. The function simply requires the data, and it will by default compare its distribution to a normal distribution. We also require the display of the line $y = x$ in order to assess the similarity of both distributions. The result is shown in figure 6.13.

```
from statsmodels.graphics.gofplots import qqplot

qqplot(residuals, line='45');    #A
```

#A Plot the Q-Q plot of the residuals. Specify the display of line *y = x*.

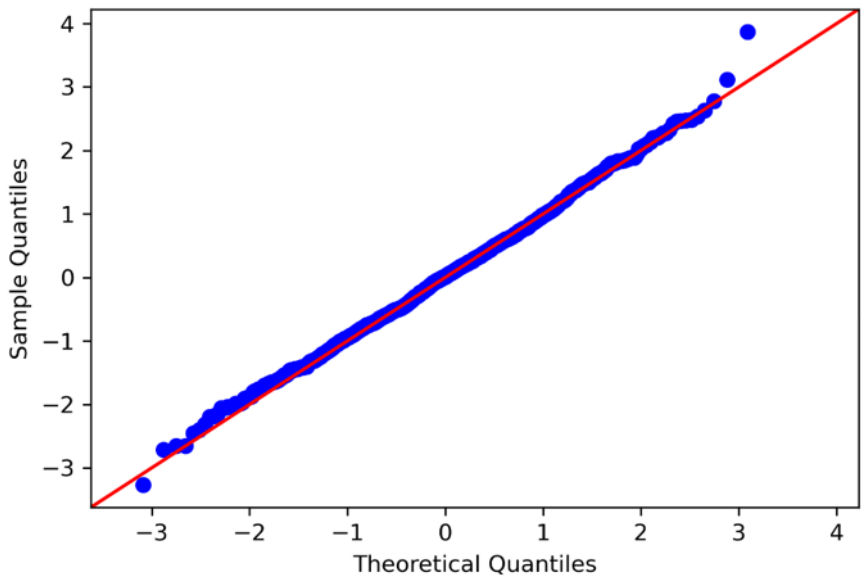


Figure 6.13 Q-Q plot of our ARMA(1,1) residuals. We can see a thick straight line lying on $y = x$. This means that our residuals are normally distributed, just like white noise.

Looking at figure 6.13, we can see a thick straight line that approximately lies on $y = x$. Therefore, from qualitative standpoint, the model's residuals seem to be normally distributed, just like white noise, which is an indication that our model fits the data well.

We extend our qualitative analysis by using the `plot_diagnostics` method. This generates a figure containing four different plots, including a Q-Q plot.

```
model_fit.plot_diagnostics(figsize=(10, 8));
```

The result is shown in figure 6.14.

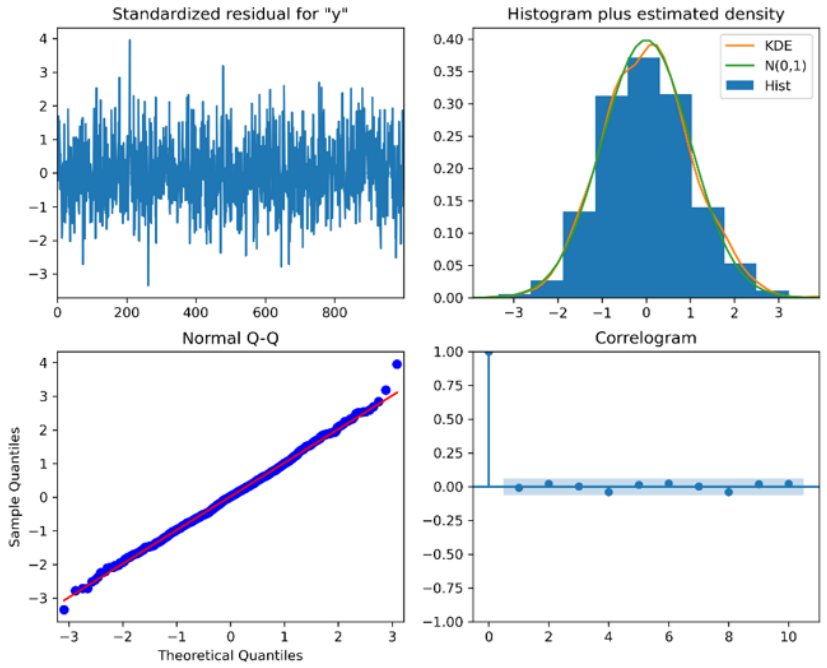


Figure 6.14 Model diagnostics from `statsmodels`. The top left plot displays the residuals. The top right plot displays the histogram of the residuals. The bottom left plot is the Q-Q plot of the residuals. The bottom right plot is the ACF plot of the residuals.

Looking at figure 6.14, we can see how `statsmodels` make it easy for us to qualitatively analyze the residuals. The top left plot shows the residuals across the entire dataset. We can see how there is not trend, and the mean seems stable over time, which is indicative of stationarity, just like white noise.

Then, the top right plot shows a histogram of the residuals. We can see the shape of a normal distribution on this plot, which again indicates that the residuals are close to white noise, as white noise is normally distributed as well.

On the bottom left, we have the Q-Q plot, which is identical to figure 6.13, and therefore leads us to the same conclusion.

Finally, the bottom right plot shows the autocorrelation function of our residuals. We see how there is only a significant peak at lag 0, and no significant coefficients otherwise. This means that the residuals are not correlated, which further supports the conclusion that they are similar to white noise, which what we expect from a good model.

The final step in residuals analysis is the application of the Ljung-Box test. This allows to quantitatively assess if our residuals are indeed uncorrelated.

We will use the `acorr_ljungbox` function from `statsmodels` to perform the Ljung-Box test on the residuals. The function takes as input the residuals as well as a list of lags. Here, we will compute the Ljung-Box statistic and p-value for 10 lags.

```
from statsmodels.stats.diagnostic import acorr_ljungbox

lbvalue, pvalue = acorr_ljungbox(residuals, np.arange(1, 11, 1))    #A

print(pvalue)    #B
```

#A Apply the Ljung-Box test on the residuals, on 10 lags.
#B Display the p-value for each lag

The resulting list of p-values shows that each is above 0.05. Therefore, at each lag, the null hypothesis cannot be rejected, meaning that the residuals are independently distributed and uncorrelated.

Therefore, we can conclude from our analysis that the residuals are similar to white noise. The Q-Q plot showed a straight line, meaning that the residuals are normally distributed. Furthermore, the Ljung-Box test shows that the residuals are uncorrelated, just like white noise. Thus, the residuals are completely random, meaning that we have a model that fits our data well.

Now, let's apply the same modeling procedure on the bandwidth dataset.

6.5 Applying the general modeling procedure

We now have a general modeling procedure that allows us to model and forecast a general ARMA(p,q) model, as outlined in figure 6.15. We applied this procedure on our simulated ARMA(1,1) process and obtained that the best model was an ARMA(1,1) model, as expected.

Now we apply the same procedure on the bandwidth dataset to obtain the best model possible for this situation.

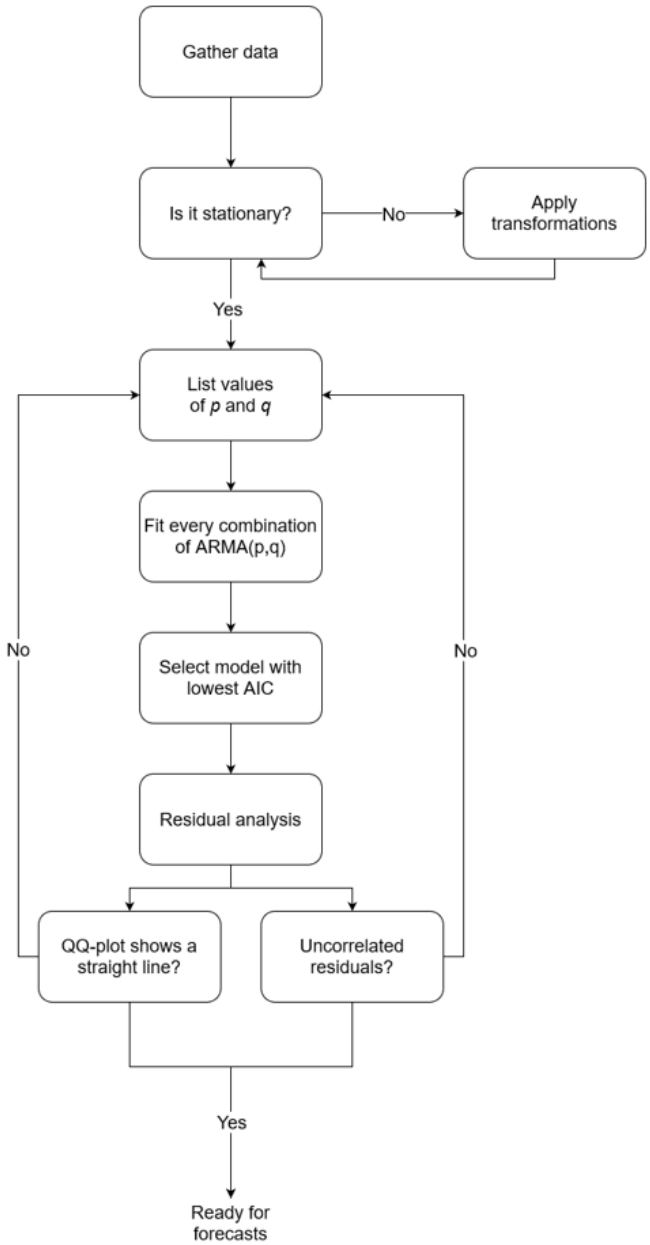


Figure 6.15 General modeling procedure for an ARMA(p,q) process.

The first step is to gather and load the data using `pandas`.

```
import pandas as pd

df = pd.read_csv('data/bandwidth.csv')
```

We can then plot our time series to look for a trend or a seasonal pattern. By now, you should be comfortable with plotting your time series. The result is shown in figure 6.16.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()

ax.plot(df.hourly_bandwidth)
ax.set_xlabel('Time')
ax.set_ylabel('Hourly bandwith usage (MBps)')

plt.xticks(
    np.arange(0, 10000, 730),
    ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec',
       '2020', 'Feb'])

fig.autofmt_xdate()
plt.tight_layout()
```

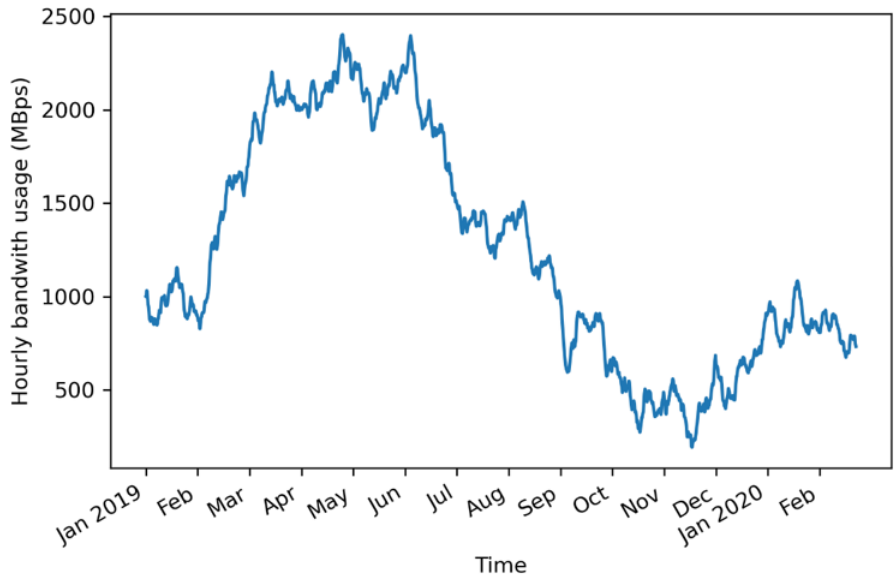


Figure 6.16 Hourly bandwidth usage in a data center since January 1st, 2019. The dataset contains 10 000 points.

With our data plotted in figure 6.16, we can see that we have no cyclical pattern in the data. However, we notice the presence of a long-term trend, meaning that our data is likely not stationary. Hence, let's apply the ADF test to verify our hypothesis. Again, we use the `adfuller` function from `statsmodels` and print out the ADF statistic and the p-value.

```
from statsmodels.tsa.stattools import adfuller

ADF_result = adfuller(df.hourly_bandwidth)

print(f'ADF Statistic: {ADF_result[0]}')
print(f'p-value: {ADF_result[1]}')
```

This prints out an ADF statistic of -0.8 and a p-value of 0.80. Therefore, we cannot reject the null hypothesis, meaning that our time series is not stationary.

We must then apply a transformation to our data in order to make it stationary. Let's apply a first-order differencing using `numpy`.

```
import numpy as np

bandwidth_diff = np.diff(df.hourly_bandwidth, n=1)
```

With this done, we can apply the ADF test again, this time on the differenced data, in order to test for stationarity.

```
ADF_result = adfuller(bandwidth_diff)

print(f'ADF Statistic: {ADF_result[0]}')
print(f'p-value: {ADF_result[1]}')
```

This returns an ADF statistic of -20.69 and a p-value of 0.0. With a large and negative ADF statistic and a p-value that is much smaller than 0.05, we can say that our differenced series is stationary.

We are now ready to start modeling our stationary process using an ARMA(p,q) model. We split our series into a training set and a test set. Since we wish to forecast the hourly bandwidth usage for the next 7 days, this means that we hold out the last 168 data points of our series for the test set. Recall that each data point is for one hour, and there are 24 hours in a day, 7 days in a week, which totals 168 hours in a week.

```
df_diff = pd.DataFrame({'bandwidth_diff': bandwidth_diff})

train = df_diff[:-168]
test = df_diff[-168:]    #A

print(len(train))
print(len(test))
```

#A There 168 hours in a week, so we assign the last 168 data points to the test set

We print out the length of the training and test set as a sanity check, and sure enough, the test set has 168 data points, and the training set has 9831 data points.

Now, let's visualize our training set and test set for both the differenced and original series. The resulting plot is shown in figure 6.17.

```
fig, (ax1, ax2) = plt.subplots(nrows=2, ncols=1, sharex=True, figsize=(10, 8))

ax1.plot(df.hourly_bandwidth)
ax1.set_xlabel('Time')
ax1.set_ylabel('Hourly bandwidth')
ax1.axvspan(9831, 10000, color='#808080', alpha=0.2)

ax2.plot(df_diff.bandwidth_diff)
ax2.set_xlabel('Time')
ax2.set_ylabel('Hourly bandwidth (diff)')
ax2.axvspan(9830, 9999, color='#808080', alpha=0.2)

plt.xticks(
    np.arange(0, 10000, 730),
    ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec',
       '2020', 'Feb'])

fig.autofmt_xdate()
plt.tight_layout()
```

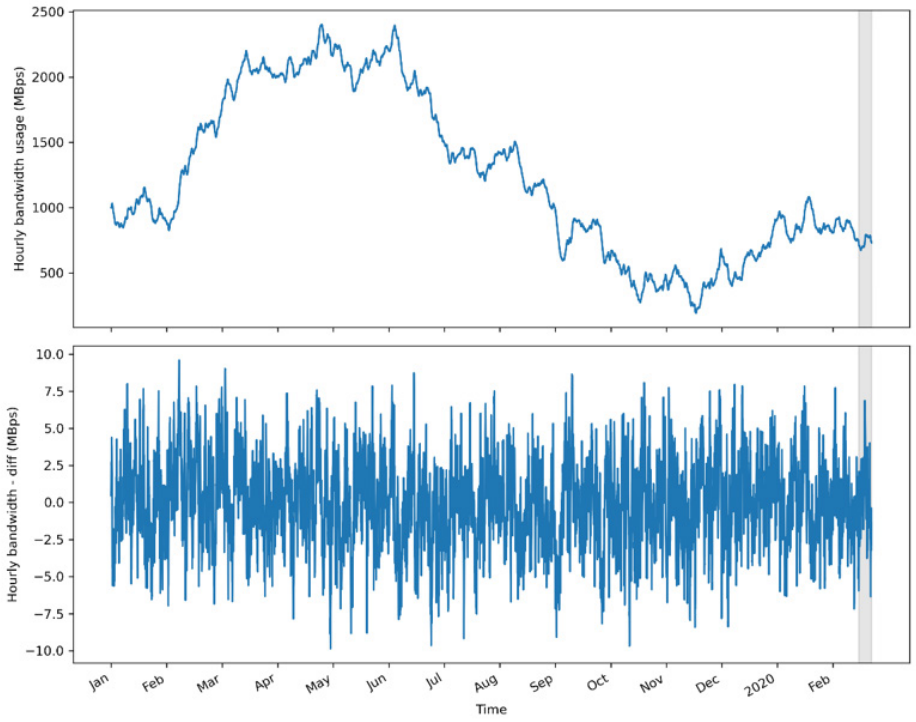


Figure 6.17 Training and test sets for the original and differenced series

With our training set ready, we can now fit different ARMA(p,q) models using the function `optimize_ARMA`, that we defined earlier. Remember that the function takes the data and the

list of unique (p,q) combinations as input. Inside the function, we initialize an empty list to store the (p,q) combination and its corresponding AIC. Then, we iterate over each (p,q) combination and fit an ARMA(p,q) model on our data. We compute the AIC and store the result. Then, we create a `DataFrame` and sort it by AIC value in ascending order, since the lower the AIC, the better the model. Our function finally outputs the ordered `DataFrame` so we can select the appropriate model. The `optimize_ARMA` function is shown in listing 6.2.

Listing 6.2 Function to fit all unique ARMA(p,q) models

```
from typing import Union
from tqdm import tqdm_notebook
from statsmodels.tsa.statespace.sarimax import SARIMAX

def optimize_ARMA(endog: Union[pd.Series, list], order_list: list) -> "[CA]"pd.DataFrame:
       #A

    results = []    #B

    for order in tqdm_notebook(order_list):    #C
        try:
            model = SARIMAX(endog, order=(order[0], 0, order[1]),
       simple_differencing=False).fit(disp=False)    #D
        except:
            continue

        aic = model.aic    #E
        results.append([order, aic])    #F

    result_df = pd.DataFrame(results)    #G
    result_df.columns = ['(p,q)', 'AIC']    #H

    #Sort in ascending order, lower AIC is better
    result_df = result_df.sort_values(by='AIC',
       "[CA]"ascending=True).reset_index(drop=True)    #I

    return result_df
```

#A The function takes as inputs the time series data, and the list of unique (p,q) combinations.
#B Initialize an empty list to store the order (p,q) and its corresponding AIC as a tuple.
#C Iterate over each unique (p,q) combination. The use of `tqdm_notebook` will display a progress bar.
#D Fit an ARMA(p,q) model using the `SARIMAX` function. We specify `simple_differencing=False` to prevent differencing. We also specify `disp=False` to avoid printing convergence messages to the console.
#E Calculate the model's AIC
#F Append the (p,q) combination and AIC as a tuple to the `results` list.
#G Store the (p,q) combination and AIC in a `DataFrame`.
#H Label the columns of your `DataFrame`.
#I Sort the `DataFrame` in ascending order of AIC values. The lower the AIC, the better the model.

Here, we will try values for p and q ranging between 0 and 3 inclusively. This means that we will 16 unique ARMA(p,q) models to our training set and select the one with the lowest AIC. Feel free to change the range of values for p and q, but keep in mind that a larger range will result in more models being fit and a longer computation time. Also, you do not need to worry about overfitting, since we are selecting our model using the Akaike's information criterion of AIC, which will prevent us from selecting a model that overfits.

```
ps = range(0, 4, 1)    #A
qs = range(0, 4, 1)    #B

order_list = list(product(ps, qs))    #C
```

#A The order p can have the values {0,1,2,3}
#B The order q can have the values {0,1,2,3}
#C Generate the unique (p,q) combinations

With this step done, we can pass in our training set and the list of unique (p,q) combination to the function `optimize_ARMA`.

```
result_df = optimize_ARMA(train['bandwidth_diff'], order_list)
result_df
```

The resulting `DataFrame` is shown in figure 6.18.

	(p,q)	AIC
0	(3, 2)	27991.063879
1	(2, 3)	27991.287509
2	(2, 2)	27991.603598
3	(3, 3)	27993.416924
4	(1, 3)	28003.349550
5	(1, 2)	28051.351401
6	(3, 1)	28071.155496
7	(3, 0)	28095.618186
8	(2, 1)	28097.250766
9	(2, 0)	28098.407664
10	(1, 1)	28172.510044
11	(1, 0)	28941.056983
12	(0, 3)	31355.802141
13	(0, 2)	33531.179284
14	(0, 1)	39402.269523
15	(0, 0)	49035.184224

Figure 6.18 `DataFrame` ordered by ascending value of AIC, resulting from fitting different ARMA(p,q) models on the differenced bandwidth dataset. Notice how the first three models all have an AIC value of 27991.

Looking at figure 6.18, we notice that the first three models all have an AIC of 27991, with only slight differences. Therefore, I would argue that the ARMA(2,2) model is the model that should be selected. Its AIC value is very close to the ARMA(3,2) and ARMA(2,3) models, while being less complex, since it has 4 parameters to be estimated instead of 5. Therefore, we select the ARMA(2,2) model to move on to the next steps, which is the analysis of its residuals.

To perform the residuals analysis, we fit an ARMA(2,2) model on our training set. Then, we use the `plot_diagnostics` method in order to study the Q-Q plot, as well as the other accompanying plots. The result is shown in figure 6.19.

```
model = SARIMAX(train['bandwidth_diff'], order=(2,0,2), "[CA]"simple_differencing=False)
model_fit = model.fit(disp=False)model_fit = best_model.fit(disp=False)
model_fit.plot_diagnostics(figsize=(10, 8));
```

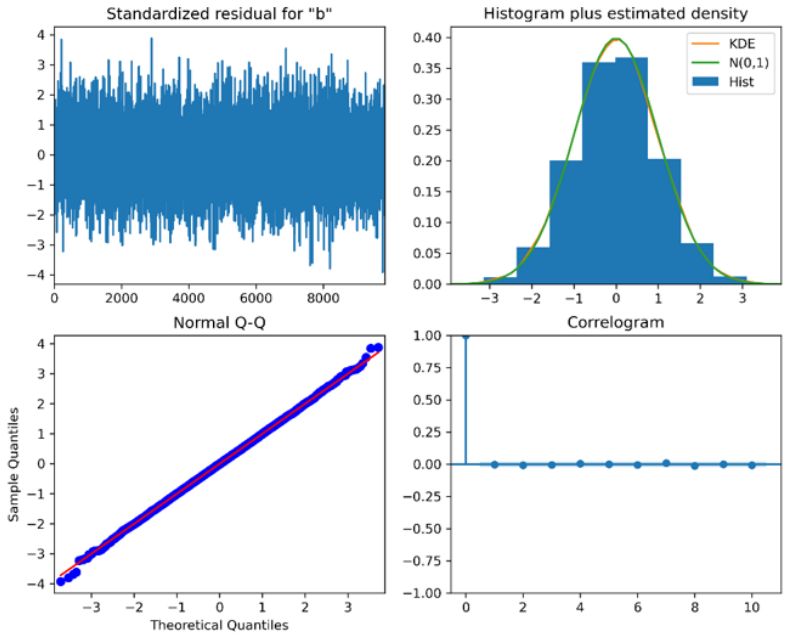


Figure 6.19 Model diagnostics from `statsmodels`. The top left plot displays the residuals. The top right plot displays the histogram of the residuals. The bottom left plot is the Q-Q plot of the residuals. The bottom right plot is the ACF plot of the residuals.

Looking at figure 6.19, we see that the top left shows no trend, and the mean seems constant over time, meaning that our residuals are likely stationary. The top right plot displays a density plot with a similar shape to that of a normal distribution. The Q-Q plot on the bottom left shows a thick straight line that is very close to $y = x$. Finally, the ACF plot at

the bottom right shows no autocorrelation after lag 0. Thus, figure 6.19 indicates that our residuals clearly resemble white noise, since they are normally distributed and uncorrelated.

Our last step is to run the Ljung-Box test on the residuals for the first 10 lags. If the returned p-values exceed 0.05, then we cannot reject the null hypothesis, which means that our residuals are uncorrelated and independently distributed, just like white noise.

```
residuals = model_fit.resid

lbvalue, pvalue = acorr_ljungbox(residuals, np.arange(1, 11, 1))

print(pvalue)
```

The returned p-values all exceed 0.05. Therefore, we can conclude that our residuals are indeed uncorrelated. Our ARMA(2,2) model has passed all checks on the residuals analysis, and we are ready to use this model to forecast bandwidth usage.

6.6 Forecasting bandwidth usage

In the previous section, we applied the general modeling procedure on the bandwidth dataset and concluded than an ARMA(2,2) model was the best model for our data. Now, we will use the ARMA(2,2) model to forecast the hourly bandwidth over 7 days.

We will reuse the `rolling_forecast` function that we defined and used in chapters 4 and 5, as shown in listing 6.3. Recall that this function allows to forecast a few timesteps at a time, until we have forecasts for the entire horizon. This time, of course, we fit an ARMA(2,2) model to our differenced data. Also, we compare the model's performance to two benchmarks: the mean and the last known value. Thus, we can make sure that an ARMA(2,2) model performs better than naïve forecasting methods.

Listing 6.3 A function to perform rolling forecast on a horizon

```
def rolling_forecast(df: pd.DataFrame, train_len: int, horizon: int, window: int, method:
       str) -> list:

    total_len = train_len + horizon
    end_idx = train_len

    if method == 'mean':
        pred_mean = []

        for i in range(train_len, total_len, window):
            mean = np.mean(df[:i].values)
            pred_mean.extend(mean for _ in range(window))

        return pred_mean

    elif method == 'last':
        pred_last_value = []

        for i in range(train_len, total_len, window):
            last_value = df[:i].iloc[-1].values[0]
            pred_last_value.extend(last_value for _ in range(window))

        return pred_last_value

    elif method == 'ARMA':
        pred_ARMA = []

        for i in range(train_len, total_len, window):
            model = SARIMAX(df[:i], order=(2,0,2))    #A
            res = model.fit(disp=False)
            predictions = res.get_prediction(0, i + window - 1)
            oos_pred = predictions.predicted_mean.iloc[-window:]
            pred_ARMA.extend(oos_pred)

        return pred_ARMA
```

#A The order specifies an ARMA(2,2) model

With `recursive_forecast` defined, we can now use it to evaluate the performance of the different forecasting methods. We first create a `DataFrame` to hold the actual values of the test set, as well as the predictions from the different methods. Then, we specify size of the training set, and the size of the test set. We will predict two steps at a time, because we have an ARMA(2,2) model, meaning that there is a MA(2) component. We know, from chapter 4, that predicting beyond q steps into the future with a MA(q) model will simply return the mean, and so the predictions will remain flat. We therefore avoid this situation by setting the window to 2. We then forecast on the test set using the mean method, last value method, and the ARMA(2,2) model, and store each forecast in its appropriate column in `pred_df`.

```
pred_df = test.copy()

TRAIN_LEN = len(train)
HORIZON = len(test)
WINDOW = 2

pred_mean = recursive_forecast(df_diff, TRAIN_LEN, HORIZON, WINDOW, 'mean')
pred_last_value = recursive_forecast(df_diff, TRAIN_LEN, HORIZON, WINDOW, 'last')
pred_ARMA = recursive_forecast(df_diff, TRAIN_LEN, HORIZON, WINDOW, 'ARMA')

pred_df['pred_mean'] = pred_mean
pred_df['pred_last_value'] = pred_last_value
pred_df['pred_ARMA'] = pred_ARMA

pred_df.head()
```

We can plot and visualize the forecasts of each method in figure 6.20. We zoom on testing period for a better visualization.

```
fig, ax = plt.subplots()

ax.plot(df_diff.bandwidth_diff)
ax.plot(pred_df.bandwidth_diff, 'b-', label='actual')
ax.plot(pred_df.pred_mean, 'g:', label='mean')
ax.plot(pred_df.pred_last_value, 'r-.', label='last')
ax.plot(pred_df.pred_ARMA, 'k--', label='ARMA(2,2)')

ax.legend(loc=2)

ax.set_xlabel('Time')
ax.set_ylabel('Hourly bandwidth (diff)')

ax.axvspan(9830, 9999, color='#808080', alpha=0.2)    #A

ax.set_xlim(9800, 9999)    #B

plt.xticks(
    [9802, 9850, 9898, 9946, 9994],
    ['2020-02-13', '2020-02-15', '2020-02-17', '2020-02-19', '2020-02-21'])

fig.autofmt_xdate()
plt.tight_layout()
```

#A Assign a gray background for the testing period
#B Zoom in on the testing period

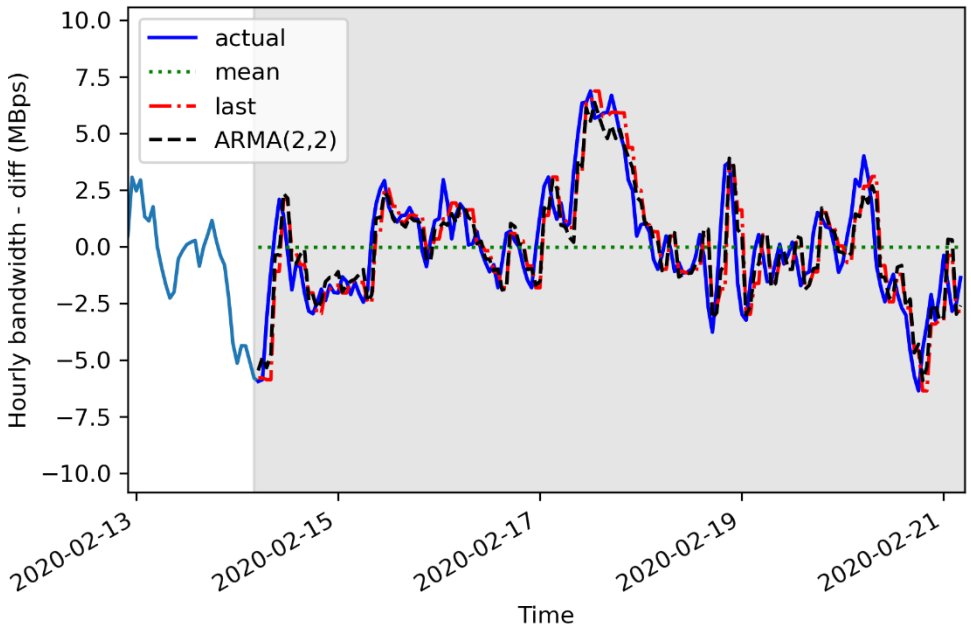


Figure 6.20 Forecasts of the differenced hourly bandwidth usage using the mean, the last known value, and an ARMA(2,2) model. We can how the ARMA(2,2) forecasts and last known value forecasts almost coincide with the actual values of the test set.

Looking at figure 6.20, we can see that the ARMA(2,2) forecasts shown as a dashed line almost coincide with the actual values of the test set. The same can be said of the forecasts from the last known value method, shown as a dashed and dotted line. Of course, the forecasts using the mean, shown as a dotted line, are mostly flat over the testing period.

We now measure the mean squared error, or MSE, to evaluate the performance of each model. The model with the lowest MSE is the best performing model.

```
mse_mean = mean_squared_error(pred_df.bandwidth_diff, pred_df.pred_mean)
mse_last = mean_squared_error(pred_df.bandwidth_diff, pred_df.pred_last_value)
mse_ARMA = mean_squared_error(pred_df.bandwidth_diff, pred_df.pred_ARMA)

print(mse_mean, mse_last, mse_ARMA)
```

This return a MSE of 6.3 for the mean method, 2.2 for the last value method, and 1.8 for the ARMA(2,2) model. Therefore, the ARMA(2,2) outperforms the benchmarks, meaning that we have a good performing model.

The final step is to reverse the transformation of our forecast in order to bring them to the same scale as our original data. Remember that we differenced the original data to make

it stationary. The ARMA(2,2) model was then applied on the stationary dataset and produced forecasts that are differenced.

To reverse the differencing transformation, we apply a cumulative sum, just as we did in chapters 4 and 5.

```
df['pred_bandwidth'] = pd.Series()
df['pred_bandwidth'][9832:] = df['hourly_bandwidth'].iloc[9832] +
       “[CA]”pred_df['pred_ARMA'].cumsum()
```

We can plot the forecasts on the original scale of the data in figure 6.21.

```
fig, ax = plt.subplots()

ax.plot(df.hourly_bandwidth)
ax.plot(df.hourly_bandwidth, 'b-', label='actual')
ax.plot(df.pred_bandwidth, 'k--', label='ARMA(2,2)')

ax.legend(loc=2)

ax.set_xlabel('Time')
ax.set_ylabel('Hourly bandwith usage (MBps)')

ax.axvspan(9831, 10000, color='#808080', alpha=0.2)

ax.set_xlim(9800, 9999)

plt.xticks(
    [9802, 9850, 9898, 9946, 9994],
    ['2020-02-13', '2020-02-15', '2020-02-17', '2020-02-19', '2020-02-21'])

fig.autofmt_xdate()
plt.tight_layout()
```

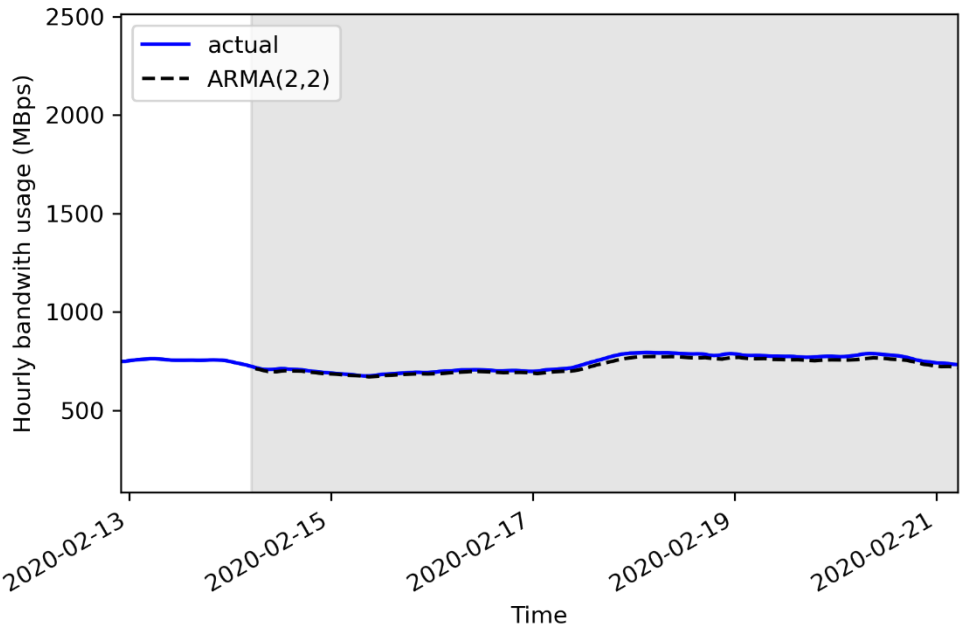


Figure 6.21 Undifferenced predictions of the hourly bandwidth usage. Notice how the dashed line representing our predictions almost coincide with the solid line representing the actual values. This means that our predictions are very close to the actual values, indicating a performant model.

Looking at figure 6.21, our forecasts shown as a dashed line are following closely the actual values of the test set, and both lines almost coincide.

We can measure the mean absolute error, or MAE, of the undifferenced ARMA(2,2) predictions to understand how far apart the predictions are from the actual values. We choose the MAE simply because it is easy to interpret.

```
mae_ARMA_undiff = mean_absolute_error(df.hourly_bandwidth[9832:],
       “[CA]”df.pred_bandwidth[9832:])

print(mae_ARMA_undiff)
```

This returns a MAE of 14, meaning that on average, our forecasts are 14 MBps above or below the actual bandwidth usage.

6.7 Next steps

In this chapter, we covered the ARMA(p,q) model and how it effectively combines an AR(p) model with a MA(q) model to model and forecast more complex time series.

This required us to define an entirely new modeling procedure that can does not rely on the qualitative study of the ACF and PACF plots. Instead, we fit many ARMA(p,q) models with different (p,q) combinations, and select the model with the lowest AIC. Then, we analyze the model's residuals to make sure that their properties is similar to white noise: normally distributed, stationary, and uncorrelated This analysis is both qualitative, as we can study the Q-Q plot to evaluate if the residuals are normally distributed, as well as quantitative, since we apply the Ljung-Box test to determine if the residuals are correlated or not. If the model's residuals have the properties of a random variable, like white noise, then the model is used for forecasting.

Up until now, we have covered different models for stationary time series, mainly the MA(q) model, AR(p) model, and ARMA(p,q) model. Each model required us to transform our data to make it stationary before we could forecast. Furthermore, we had to reverse the transformation on our forecast, in order to obtain predictions in the original scale of the data.

Now, there is a way to model non-stationary time series, without having to transform them and reverse the transformation on the predictions. Specifically, we can model *integrated* time series using the **autoregressive integrated moving average** model or **ARIMA(p,d,q)**. This will be the subject of the next chapter.

6.8 Exercises

6.8.1 Easy: Make predictions on the simulated ARMA(1,1) process

- Reusing the simulated ARMA(1,1) process, split it into a training and test set. Assign 80% of the data to the training set, and the remaining 20% to the test set.
- Use the `recursive_forecast` function to make predictions using the ARMA(1,1) model, the mean method, and the last know value method.
- Plot your forecasts.
- Evaluate each method's performance using the MSE. Which method performed best?

6.8.2 Medium: Simulate an ARMA(2,2) process and make forecasts

- Simulate a stationary ARMA(2,2) process. Use the `ArmaProcess` function from `statsmodels` and simulate:

$$y_t = 0.33y_{t-1} + 0.5y_{t-2} + 0.9\epsilon_{t-1} + 0.3\epsilon_{t-2}$$

Simulate 10 000 samples.

```
from statsmodels.tsa.arima_process import ArmaProcess
import numpy as np

np.random.seed(42)    #A

ma2 = np.array([1, 0.9, 0.3])
ar2 = np.array([1, -0.33, -0.5])

ARMA_2_2 = ArmaProcess(ar2, ma2).generate_sample(nsample=10000)
```

#A Set the seed for reproducibility. There is randomness every time you generate samples. By setting the seed, you make sure that the same values are obtained no matter how many times the code is run. Change the seed if you want to experiment with different values.

- Plot your simulated process.
- Test for stationarity using the ADF test.
- Define a range of values for p and q and generate all unique combinations of orders (p,q).
- Use the function `optimize_ARMA` to fit all unique ARMA(p,q) models and select the one with the lowest AIC. Is the ARMA(2,2) model the one with the lowest AIC?
- Select the best model according to the AIC and store the residuals in a variable called `residuals`.
- Perform a qualitative analysis of the residuals with the `plot_diagnostics` method. Does the Q-Q plot show a straight line that lies on y = x? Does the correlogram show significant coefficients?
- Perform a quantitative analysis of the residuals by applying the Ljung-Box test on the first 10 lags. Are all returned p-values above 0.05? Are the residuals correlated or not?
- Split your data into a training and test set. The first 80% goes for training, and the remaining 20% goes for testing.
- Use the `recursive_forecast` function to make predictions using the selected ARMA(p,q) model, the mean method, and the last know value method.
- Plot your forecasts.
- Evaluate each method's performance using the MSE. Which method performed best?

6.9 Summary

- The autoregressive moving average model, denoted as ARMA(p,q), is the combination of the autoregressive model AR(p), and the moving average model MA(q).
- An ARMA(p,q) process will display a decaying pattern or a sinusoidal pattern on both the ACF and PACF plots. Therefore, they cannot be used to estimate the orders p and q.
- The general modeling procedure does not rely on the ACF and PACF plots. Instead, we fit many ARMA(p,q) models and perform model selection and residuals analysis.
- Model selection is done with the Akaike's Information Criterion or AIC. It quantifies the information loss of a model, and it is related to the number of parameters in a model and its goodness of fit. The lower the AIC, the better the model.

- The AIC is relative measure of quality. It returns the best model among other models. For an absolute measure of quality, we perform residuals analysis.
- Residuals of a good model must approximate white noise, meaning that they must be uncorrelated, normally distributed, and independent.
- The Q-Q plot is a graphical tool to compare two distributions. We use it to compare the distribution of the residuals against a theoretical normal distribution. If the plot shows a straight line that lies on $y=x$, then both distributions are similar. Otherwise, it means that the residuals are not normally distributed.
- The Ljung-Box test allows us to determine if the residuals are correlated or not. The null hypothesis states that the data is independently distributed and uncorrelated. If the returned p-values are larger than 0.05, then we cannot reject the null hypothesis, meaning that the residuals are uncorrelated, just like white noise.

7

Forecasting non-stationary time series

This chapter covers

- Examining the autoregressive integrated moving average model or ARIMA(p,d,q)
- Applying the general modeling procedure for non-stationary time series
- Forecasting using the ARIMA(p,d,q) model

In chapters 4, 5, and 6, we covered the moving average model or MA(q), the autoregressive model or AR(p), and the ARMA model or ARMA(p,q). We saw how these models can only be used for stationary time series, which required us to apply transformations, mainly differencing, and testing for stationarity using the ADF test. In the examples that we covered, the forecasts from each model returned differenced values, which required us to reverse this transformation in order to bring the values back to the scale of the original data.

Now, we introduce another component to the ARMA(p,q) model to forecast non-stationary time series. This component is the integration order, which is denoted by the variable d. This leads us to the **AutoRegressive Integrated Moving Average** model **or ARIMA(p,d,q)**. Using this model, we can take into account non-stationary time series and avoid the steps of modeling on differenced data and having to *undifference* the forecasts.

In this chapter, we define the ARIMA(p,d,q) model and the order of integration d. Then, we add a step to our general modeling procedure. In figure, 7.1, we can see the general modeling procedure as defined in chapter 6. Now, we must add a step to determine the order of integration to use it in the ARIMA(p,d,q) model.

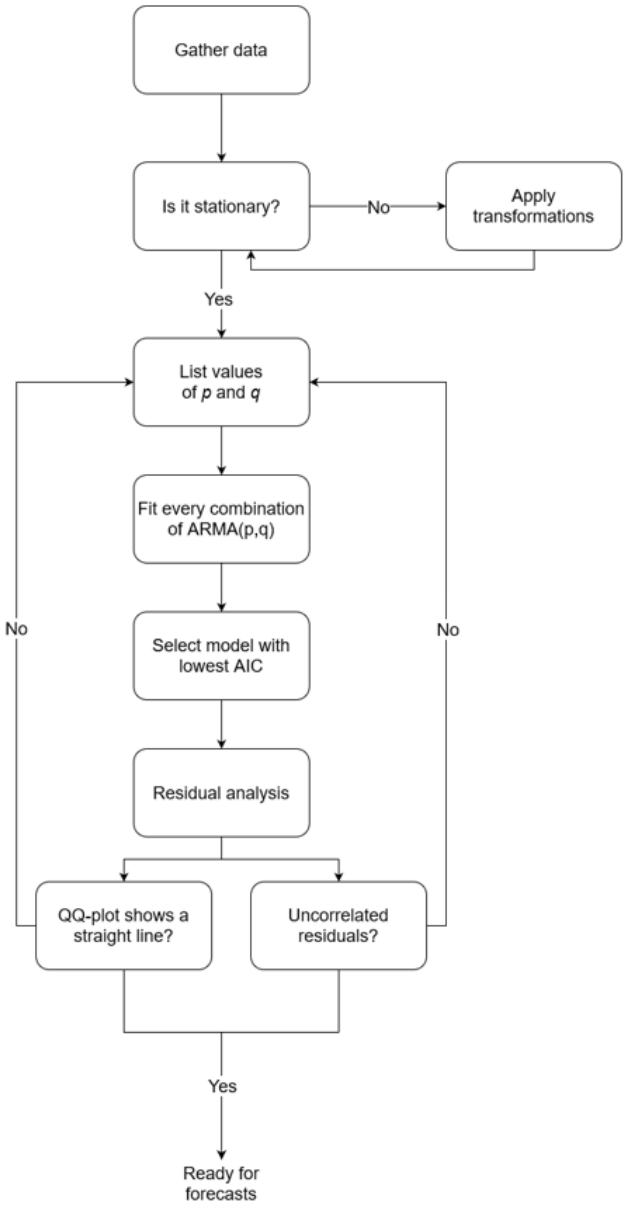


Figure 7.1 General modeling procedure using an ARMA(p,q) model. We will later add another step to this procedure in order to accommodate the ARIMA(p,d,q) model.

Then, we apply our modified procedure to forecast a non-stationary time series, meaning that the series has a trend, or its variance is not constant over time. Specifically, we revisit the dataset of Johnson & Johnson's quarterly earnings per share (EPS) between 1960 and 1980 that we first studied in both chapter 1 and 2. The series is shown in figure 7.2. We apply the ARIMA(p,d,q) model to forecast the quarterly EPS for the following year.

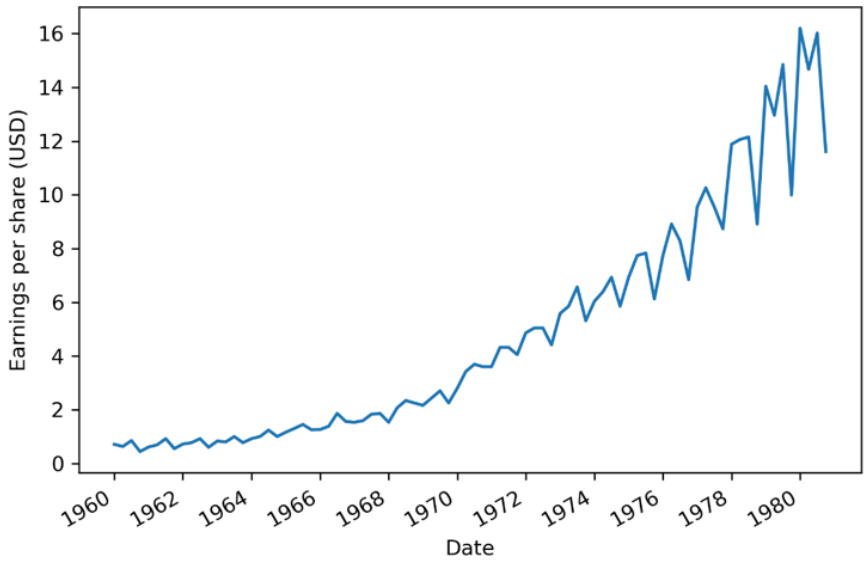


Figure 7.2 Quarterly earnings per share (EPS) of Johnson & Johnson between 1960 and 1980. Note that we worked with the same dataset in chapters 1 and 2.

7.1 Defining the autoregressive integrated moving average model

An autoregressive integrated moving average process is the combination of an autoregressive process AR(p), integration I(d), and the moving average process MA(q).

Just like the ARMA process, the ARIMA process states that the present value dependent on past values, coming from the AR(p) portion, and past errors, coming from the MA(q) portion. However, instead of using the original series, denoted as y_t, the ARIMA process uses the differenced series denoted as y'_t. Note that y'_t can represent a series that was differenced more than once.

Therefore, the mathematical expression of the ARIMA(p,d,q) process defines that the present value of the differenced series y'_t is equal to the sum of a constant C, past values of the differenced series $\varphi_p y'_{t-p}$, the mean of the differenced series μ, past error terms $\theta_q \varepsilon_{t-q}$, and a current error term ε_t, as shown in equation 7.1.

$$y'_t = C + \phi y'_{t-1} + \ldots + \phi_p y'_{t-p} + \mu + \theta_1 \epsilon'_{t-1} + \ldots + \theta_q \epsilon'_{t-q} + \epsilon_t$$

Equation 7.1

Just like in the ARMA process, the order p determines how many lagged values of the series are included in the model, while the order q determines how many lagged error terms are included in the model. However, looking at equation 7.1, we notice that there is not order d explicitly displayed.

Here, the order d is defined as the order of integration. Integration is simply the reverse of differencing. Therefore, from now on, we will use the term *integrate* instead of *undifference*. The order of integration is then equal to the number of times a series was differenced to become stationary.

If we difference a series once and it becomes stationary, then $d = 1$. If a series is differenced twice to become stationary, then $d = 2$.

Autoregressive integrated moving average model

An autoregressive integrated moving average process is the combination of an AR(p) and a MA(q) process, but in terms of the differenced series.

It is denoted as ARIMA(p,d,q), where p is the order of the AR(p) process, d is the order of integration, and q is the order of the MA(q) process.

Integration is the reverse of differencing.

The order of integration d is equal to the number of times the series was differenced to be rendered stationary.

The general equation of ARIMA(p,d,q) process is:

$$y'_t = C + \phi y'_{t-1} + \ldots + \phi_p y'_{t-p} + \mu + \theta_1 \epsilon'_{t-1} + \ldots + \theta_q \epsilon'_{t-q} + \epsilon_t$$

Note that y'_t represents the differenced series, and it may have been differenced more than once.

A time series that can be rendered stationary by applying differencing is said to be an *integrated* series. In the presence of a non-stationary integrated time series, we can then use the ARIMA(p,d,q) model to produce forecasts.

Thus, in simple terms, the ARIMA model is simply an ARMA model that can be applied on non-stationary time series. Whereas the ARMA(p,q) model requires the series to be stationary before fitting an ARMA(p,q) model, the ARIMA(p,d,q) model can be used on non-stationary series. We must simply find the order of integration d, which corresponds to the minimum number of times a series must be differenced to become stationary.

Therefore, we must add the step of finding the order of integration to our general modeling procedure before we apply it to forecast the quarterly EPS of Johnson & Johnson.

7.2 Modifying the general modeling procedure to account for non-stationary series

In chapter 6, built a general modeling procedure that allowed us to model more complex time series, meaning that the series has both an autoregressive and a moving average component.

This procedure involves fitting many ARMA(p,q) models and selecting the one with the lowest AIC. Then, we must study the model's residuals to verify that they resemble white noise. If that is the case, then the model can be used for forecasting. We can visualize the general modeling procedure in its present state in figure 7.3.

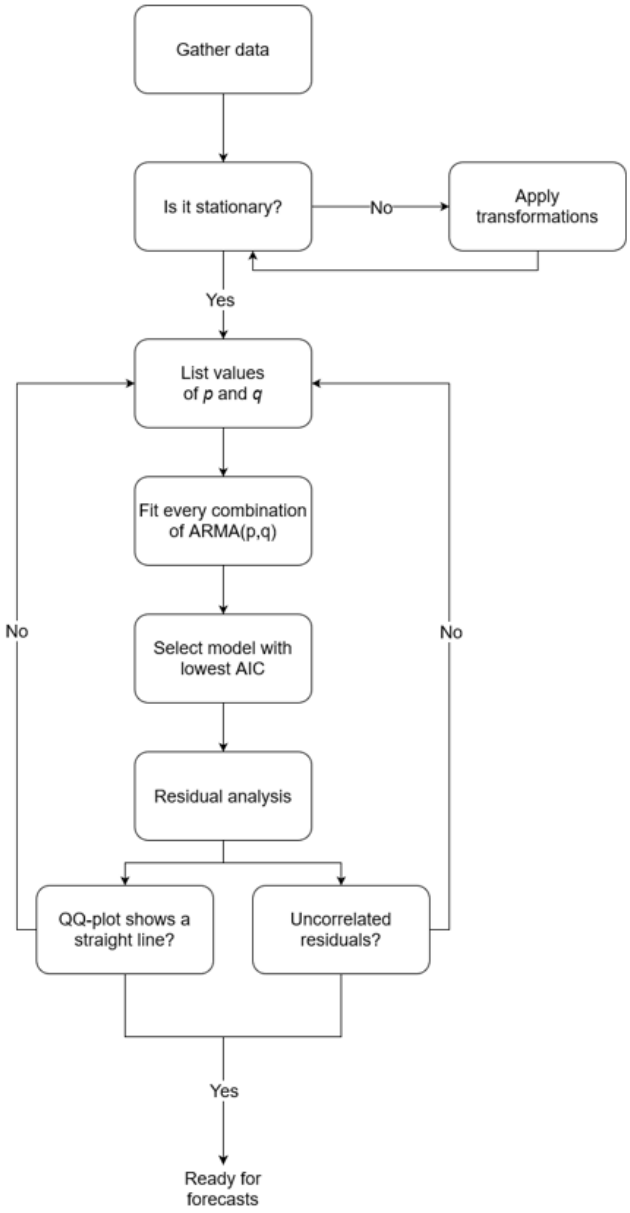


Figure 7.3 General modeling procedure using an ARMA(p,q) model. Now, we must adapt it to apply an ARIMA(p,d,q) model, allowing us to work with non-stationary time series.

The next iteration of the general modeling procedure is to include a step to determine the order of integration d. That way, we can apply the same procedure but using an ARIMA(p,d,q) model, which allows us to forecast non-stationary time series.

From the previous section, we know that the order of integration d is simply the minimum number of times a series was differenced to become stationary. Therefore, if a series is stationary after being differenced once, then $d = 1$. If it is stationary after being differenced twice, then $d = 2$. From experience, a time series rarely needs to be differenced more than twice to become stationary.

Hence, we add a step such that when transformations are applied to the series, we set the value of d to the number of times the series was differenced. Then, instead of fitting many ARMA(p,q) models, we fit many ARIMA(p,d,q) models. The rest of procedure remains the same, as we still use the AIC to select the best model and study its residuals. The resulting procedure is shown in figure 7.4.

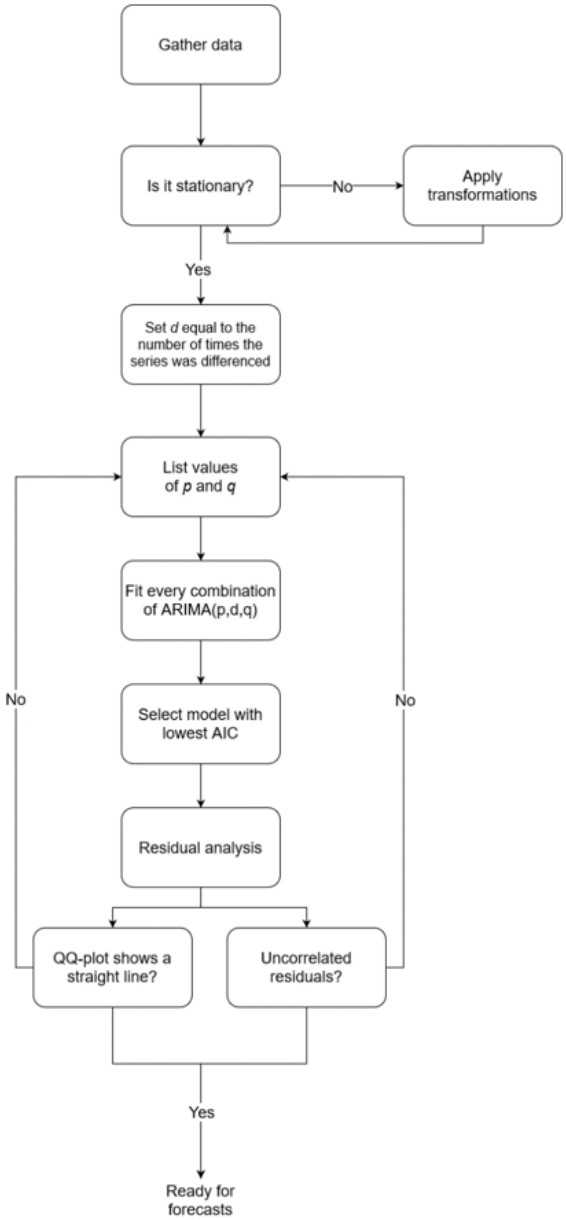


Figure 7.4 General modeling procedure using the ARIMA(p,d,q) model. Notice the addition of a step where we specify the parameter d of the ARIMA(p,d,q) model. Here, d is simply the minimum number of times a series must be differenced to become stationary.

Looking at figure 7.4, we now see the additional step of setting d to the number of times the series was differenced to become stationary.

Note that in the case where $d = 0$, then it is equivalent to an ARMA(p,q) model. This also means that the series did not need to be differenced to be stationary.

It must also be specified that the ARMA(p,q) model can only be applied on a stationary series, whereas the ARIMA(p,d,q) model can be applied on a series that was not differenced.

Let's apply our new general modeling procedure to forecast the quarterly earnings per share of Johnson & Jonhson.

7.3 Forecasting a non-stationary times series

We are now going to apply the general modeling procedure displayed in figure 7.4 to forecast the quarterly earnings per share (EPS) of Johnson & Jonhson. We use the same dataset that was introduced in chapters 1 and 2. Here, we wish to forecast next year's quarterly EPS, meaning that we must forecast four timesteps into the future, since there are four quarters in a year. The dataset covers the period between 1960 and 1980.

As always, the first step is to collect our data. Here, it is done for us, so we can simply load it and display the series. The result is shown in figure 7.5.

```
df = pd.read_csv('data/jj.csv')

fig, ax = plt.subplots()

ax.plot(df.date, df.data)
ax.set_xlabel('Date')
ax.set_ylabel('Earnings per share (USD)')

plt.xticks(np.arange(0, 81, 8), [1960, 1962, 1964, 1966, 1968, 1970, 1972, 1974, 1976,
       "[CA]"1978, 1980])

fig.autofmt_xdate()
plt.tight_layout()
```

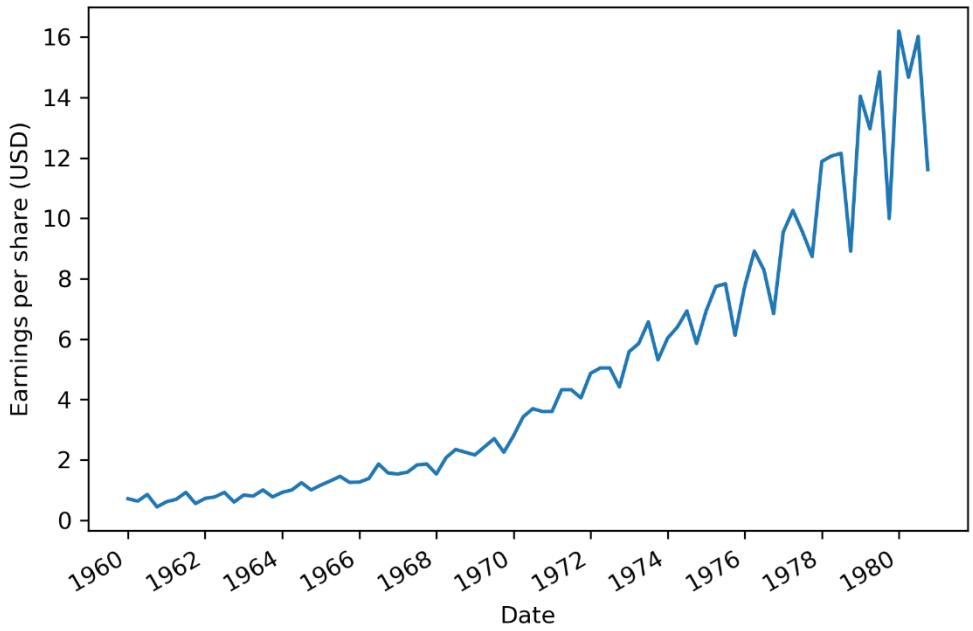


Figure 7.5 Quarterly earnings per share (EPS) of Johnson & Johnson between 1960 and 1980. Note that we worked with the same dataset in chapters 1 and 2.

Then, following the procedure, we must check the if the data is stationary. Looking at figure 7.5, we notice the presence of a positive trend, as the quarterly EPS tends to increase over time. Nevertheless, we apply the Augmented Dickey-Fuller test, or ADF test to determine if it is stationary or not. By now, you should be very comfortable with these steps, so they will be accompanied by minimal comments.

```
ad_fuller_result = adfuller(df.data)

print(f'ADF Statistic: {ad_fuller_result[0]}')
print(f'p-value: {ad_fuller_result[1]}')
```

This block of code returns an ADF statistic of 2.74 with a p-value of 1.0. Since the ADF statistic is not a large negative number, and the p-value is larger than 0.05, we cannot reject the null hypothesis, meaning that our series is not stationary.

Now, we need to determine how many times it must be differenced to become stationary. This will then set the order of integration d.

Therefore, we apply a first-order differencing and test for stationarity.

```
eps_diff = np.diff(df.data, n=1)    #A

ad_fuller_result = adfuller(eps_diff)    #B

print(f'ADF Statistic: {ad_fuller_result[0]}')
print(f'p-value: {ad_fuller_result[1]}')
```

#A Apply first-order differencing
#B Test for stationarity

This results in an ADF statistic of -0.41 and a p-value of 0.9. Again, the ADF statistic is not a large negative number, and the p-value is larger than 0.05. Therefore, we cannot reject the null hypothesis and we must conclude that after a first-order differencing, the series is not stationary.

Now, let's try differencing again to see if the series become stationary.

```
eps_diff2 = np.diff(eps_diff, n=1)    #A

ad_fuller_result = adfuller(eps_diff2)    #B

print(f'ADF Statistic: {ad_fuller_result[0]}')
print(f'p-value: {ad_fuller_result[1]}')
```

#A Take the differenced series and difference it again
#B Test for stationarity

This results in an ADF statistic of -3.59 and a p-value of 0.006. Now that we have a p-value smaller than 0.05 and a large negative ADF statistic, we can reject the null hypothesis and conclude that our series is stationary. Therefore, it took two rounds of differencing to make our data stationary. This means that our order of integration is 2, therefore $d = 2$.

Before we move on to fitting different combinations of ARIMA(p,d,q) models, we must separate our data into a training set and a test set. Here, we hold out the last year of data for testing. This means that we will fit the model with data from 1960 to 1979 and predict the quarterly EPS in 1980 to evaluate the quality of our model against the observed values in 1980. We can visualize the testing period in figure 7.6, as the shaded area.

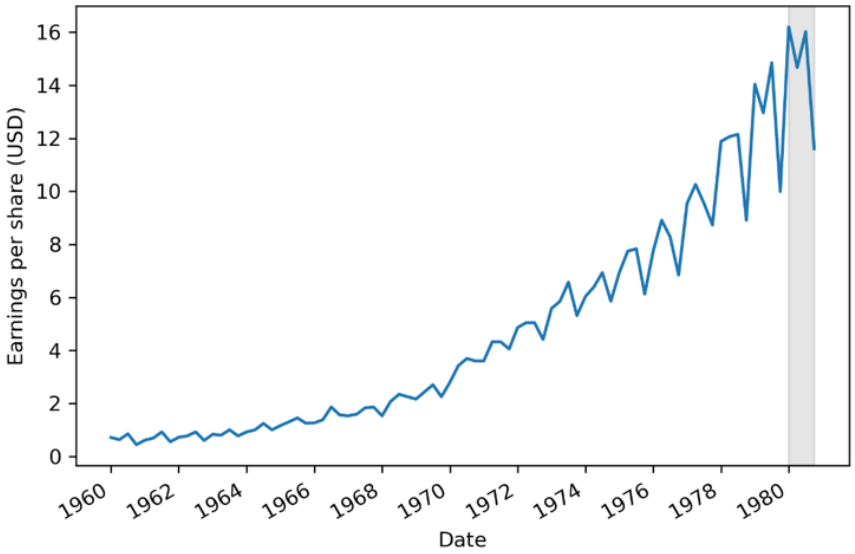


Figure 7.6 Training set and test set. The training period span the years 1960 to 1979 inclusively, while the test set is quarterly EPS reported in 1980. Note that this corresponds to the last four data points of the dataset.

Now, to fit many ARIMA(p,d,q) models, we define the `optimize_ARIMA` function. It is almost identical to the `optimize_ARMA` function that we defined in chapter 6, only this time, we add the order of integration d as an input to the function. The remaining of the function stays the same, as we fit different models and order them by ascending AIC in order to select the model with the lowest AIC. The function `optimize_ARIMA` is shown in listing 7.1.

Listing 7.1 Function to fit all unique ARIMA(p,d,q) models

```
from typing import Union
from tqdm import tqdm_notebook
from statsmodels.tsa.statespace.sarimax import SARIMAX

def optimize_ARIMA(endog: Union[pd.Series, list], order_list: list, d: int) “[CA]”->
       pd.DataFrame:    #A

    results = []    #B

    for order in tqdm_notebook(order_list):    #C
        try:
            model = SARIMAX(endog, order=(order[0], d, order[1]),
       “[CA]”simple_differencing=False).fit(disp=False)    #D
        except:
            continue
        aic = model.aic    #E
        results.append([order, aic])    #F

    result_df = pd.DataFrame(results)    #G
    result_df.columns = ['(p,q)', 'AIC']    #H

    #Sort in ascending order, lower AIC is better
    result_df = result_df.sort_values(by='AIC',
       “[CA]”ascending=True).reset_index(drop=True)    #I

    return result_df
```

#A The function takes as inputs the time series data, the list of unique (p,q) combinations, and the order of integration d.
#B Initialize an empty list to store the order (p,q) and its corresponding AIC as a tuple.
#C Iterate over each unique (p,q) combination. The use of `tqdm_notebook` will display a progress bar.
#D Fit an ARIMA(p,d,q) model using the `SARIMAX` function. We specify `simple_differencing=False` to prevent differencing. We also specify `disp=False` to avoid printing convergence messages to the console.
#E Calculate the model's AIC
#F Append the (p,q) combination and AIC as a tuple to the `results` list.
#G Store the (p,q) combination and AIC in a `DataFrame`.
#H Label the columns of your `DataFrame`.
#I Sort the `DataFrame` in ascending order of AIC values. The lower the AIC, the better the model.

With the function in place, we can define a list of possible values for the orders p and q. In this case, we try the values 0, 1, 2, and 3 for both orders and generate the list of unique (p,q) combinations.

```
from itertools import product

ps = range(0, 4, 1)    #A
qs = range(0, 4, 1)    #B
d = 2    #C

order_list = list(product(ps, qs))    #D
```

#A Create a list of possible values for p starting from 0 inclusively to 4 exclusively, with steps of 1.
#B Create a list of possible values for q starting from 0 inclusively to 4 exclusively, with steps of 1.
#C Set d to 2, as the series needed to be differenced twice to become stationary.
#D Generate a list containing all unique combination of (p,q).

Note that we do not give a range of values for the parameter d. This is because d has a very specific definition: it is the number of times a series must be differenced to become stationary. Hence, it must be set to a specific value, which in this case is 2.

Furthermore, d must be constant in order to compare models using the AIC. Varying d will change the likelihood function used in the calculation of the AIC value, and then comparing models using the AIC as a criterion is not valid anymore.

We can now run the `optimize_ARIMA` function using the training set. The function returns a `DataFrame` with the model with the lowest AIC at the top.

```
train = df.data[:-4]    #A

result_df = optimize_ARIMA(train, order_list, d)    #B
result_df    #C
```

#A The training set consists of all data points except the last four.
#B Run the `optimize_ARIMA` function to obtain the model with the lowest AIC.
#C Display the resulting `DataFrame`.

The returned DataFrame shows that a value of 3 for both p and q results in the lowest AIC. Therefore, an ARIMA(3,2,3) seems to be the most suitable for this situation. Now, we must assess the validity of the model by studying its residuals.

To do so, we fit an ARIMA(3,2,3) model on the training set and display the residuals' diagnostics using the `plot_diagnostics` method. The result is shown in figure 7.7.

```
model = SARIMAX(train, order=(3,2,3), simple_differencing=False)    #A
model_fit = model.fit(disp=False)

model_fit.plot_diagnostics(figsize=(10,8));    #B
```

#A Fit an ARIMA(3,2,3) model on the training set, since this model has the lowest AIC.
#B Display the residuals' diagnostics.

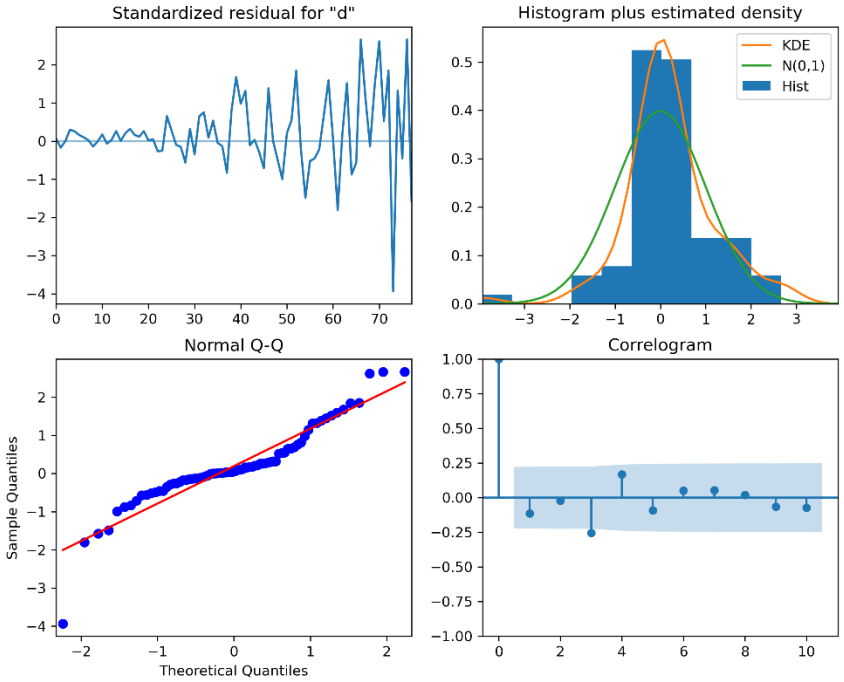


Figure 7.7 Diagnostics of the ARIMA(3,2,3) residuals. Looking at the Q-Q plot on the bottom left, we can see that it displays a fairly straight line with some deviation at the extremities.

From figure 7.7, the top left plot shows the residuals over time. While there is not trend in the residuals, the variance does not seem to be constant, which is a discrepancy in comparison to white noise. On the top right, we see the distribution of the residuals. We can see it is fairly close to a normal distribution. The Q-Q plot leads us to the same conclusion, as it displays a line that is fairly straight, meaning that the residuals' distribution is close to a normal distribution. Finally, looking at the correlogram at the bottom right, we can see that a coefficient seems to be significant at lag 3. However, since it is not preceded by any significant autocorrelation coefficients, we can assume that this is due to chance. Therefore, we can say that the correlogram shows no significant coefficients after lag 0, just like white noise.

Thus, from a qualitative standpoint, it seems that our residuals are close to white noise, which is a good sign, as it means that the model's errors are random.

The last step is to evaluate the residuals on from a quantitative standpoint. We thus apply the Ljung-Box test to determine if the residuals are correlated. We apply the test on the first 10 lags and study the p-values. If all p-values are greater than 0.05, then we cannot

reject the null hypothesis and we conclude that the residuals are not correlated, just like white noise.

```
from statsmodels.stats.diagnostic import acorr_ljungbox

residuals = model_fit.resid    #A

lbvalue, pvalue = acorr_ljungbox(residuals, np.arange(1, 11, 1))    #B

print(pvalue)
```

#A Store the model's residuals in a variable
#B Apply the Ljung-Box test on the first 10 lags

Running the Ljung-Box test on the first 10 lags of the model's residuals returns a list of p-values that are all larger than 0.05. Therefore, we do not reject the null hypothesis and conclude that the residuals are not correlated, just like white noise.

Our ARIMA(3,2,3) model has passed all checks and it can now be used for forecasting. Remember that our test set is the last four data points, corresponding to the four quarterly EPS reported in 1980. As a benchmark to our model, we will use the naïve seasonal method. This means that we take the EPS of the first quarter of 1979 and use it as a forecast for the EPS of the first quarter of 1980. Then, the EPS of the second quarter of 1979 is used as a forecast for the EPS of the second quarter of 1980, and so on. Remember that we need a benchmark, or a baseline model, when modeling to determine if the model we develop is better than a naïve method. The performance of a model must always be assessed relative to a baseline model.

```
test = df.iloc[-4:]    #A

test['naive_seasonal'] = df['data'].iloc[76:80].values    #B
```

#A The test set corresponds to the last four data points.
#B The naïve seasonal forecast is implemented by selecting the quarterly EPS reported in 1979 and using the same values as a forecast for the year 1980.

With our baseline in place, we now make forecasts using the ARIMA(3,2,3) model and store the result in the `ARIMA_pred` column.

```
ARIMA_pred = model_fit.get_prediction(80, 83).predicted_mean    #A

test['ARIMA_pred'] = ARIMA_pred    #B
```

#A Get the predicted values for the year 1980.
#B Assign the forecasts to the `ARIMA_pred` column.

Let's visualize our forecasts to see how close the predictions from each method are from the observed values. The resulting plot is shown in figure 7.8.

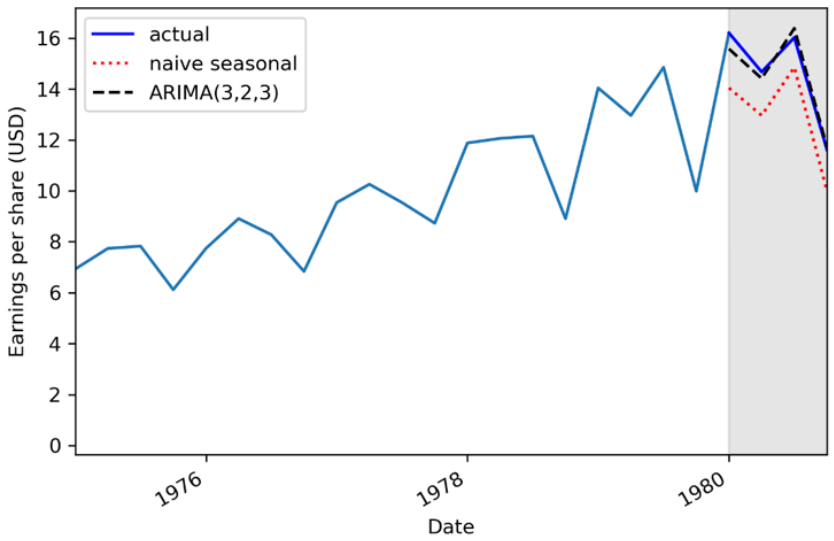


Figure 7.8 Forecasts of the quarterly EPS of Johnson & Johnson in 1980. We can see that the predictions coming from the ARIMA(3,2,3) model, shown as a dashed line, almost perfectly overlap the observed data in 1980.

From figure 7.8, we see the naïve seasonal forecast as a dotted line, and the ARIMA(3,2,3) forecasts as a dashed line. We notice that the dashed line almost overlaps the observed data in the 1980, meaning that the ARIMA(3,2,3) model predicted the quarterly EPS with a very small error.

We quantify that error by measuring the mean absolute percentage error or MAPE and display the metric for each forecasting method in a bar plot, as shown in figure 7.9.

```
def mape(y_true, y_pred):    #A
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100

mape_naive_seasonal = mape(test.data, test.naive_seasonal)    #B
mape_ARIMA = mape(test.data, test.ARIMA_pred)    #C

fig, ax = plt.subplots()

x = ['naive seasonal', 'ARIMA(3,2,3)']
y = [mape_naive_seasonal, mape_ARIMA]

ax.bar(x, y, width=0.4)
ax.set_xlabel('Models')
ax.set_ylabel('MAPE (%)')
ax.set_ylim(0, 15)

for index, value in enumerate(y):
    plt.text(x=index, y=value + 1, s=str(round(value,2)), ha='center')

plt.tight_layout()
```

```
#A Define a function to compute the MAPE
#B Compute the MAPE for the naïve seasonal method
#C Compute the MAPE for the ARIMA(3,2,3) model
```

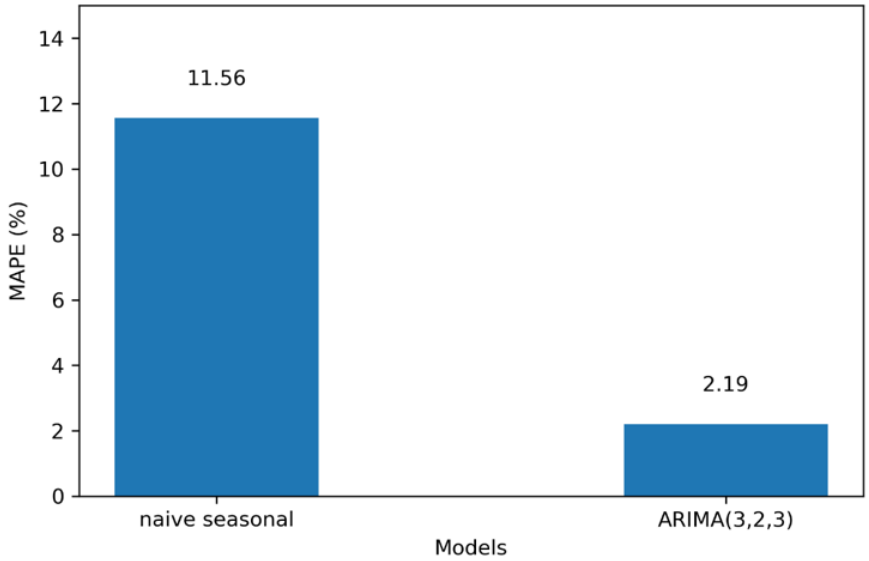


Figure 7.9 MAPE for both forecasting methods. We can see that the ARIMA model has an error metric 5x lower than the baseline.

From figure 7.9, we that the MAPE for the naïve seasonal forecast is 11.56%, while the MAPE for the ARIMA(3,2,3) model is 2.19%, which roughly 5x lower than the benchmark. This means that our predictions are on average 2.19% off from the actual value. The ARIMA(3,2,3) is without a doubt a better model than the naïve seasonal method.

7.4 Next steps

In this chapter, we covered the ARIMA(p,d,q) model which allows to model and forecast non-stationary time series.

The order of integration d defines how many times a series must be differenced to become stationary. This parameter then allows us to fit the model on the original series and get forecast in the same scale, unlike the ARMA(p,q) model that required the series to be stationary for the model to be applied and required us to reverse the transformations on the forecasts.

To apply the ARIMA(p,d,q) model, we simply added an extra step to our general modeling procedure, which simply involves finding the value for the order of integration, which corresponds to the minimum number of times a series must be differenced to become stationary.

Now, we can add another layer to the ARIMA(p,d,q) model that allows us to consider yet another property of time series: seasonality. We have studied the Johnson & Johnson dataset enough times to realize that there are clear cyclical patterns in the series. To integrate the seasonality of a series in a model, we must use the **seasonal autoregressive integrated moving average** model or **SARIMA(p,d,q)(P,D,Q)**$_m$ model. This will be the subject of the next chapter.

7.5 Exercises

7.5.1 Medium: Apply the ARIMA(p,d,q) model on the datasets from chapters 4, 5 or 6.

In chapters 4, 5 and 6, non-stationary time series were introduced to understand how to apply the MA(q) model, AR(p) model and ARMA(p,q) model. In each chapter, we transformed the series to make it stationary, fit the mode, made forecasts, and had to reverse the transformation on the forecasts to bring them back to the original scale of the data.

Now that we know how to account for non-stationary time series, revisit each dataset and apply the ARIMA(p,d,q) model. For each dataset:

- Apply the general modeling procedure.
- Is an ARIMA(0,1,2) model suitable for the dataset in chapter 4?
- Is an ARIMA(3,1,0) model suitable for the dataset in chapter 5?
- Is an ARIMA(2,1,2) model suitable for the dataset in chapter 6?

7.6 Summary

- The autoregressive integrated moving average model, denoted as ARIMA(p,d,q), is the combination of the autoregressive model AR(p), the order of integration d, and the moving average model MA(q).
- The ARIMA(p,d,q) model can be applied on non-stationary time series and has the added advantage of returning forecasts in same scale as the original series.
- The order of integration d is equal to the minimum number of times a series must be differenced to become stationary.
- An ARIMA(p,0,d) model is equivalent to an ARMA(p,q) model.

8

Accounting for seasonality

This chapter covers

- Examining the seasonal autoregressive integrated moving average model or SARIMA$(p,d,q)(P,D,Q)_m$
- Analyzing seasonal patterns in a time series
- Forecasting using the SARIMA$(p,d,q)(P,D,Q)_m$ model

In the previous chapter, we covered the autoregressive integrated moving average model or ARIMA(p,d,q), which allows to model non-stationary time series. Now, we add another layer of complexity to the ARIMA model to include seasonal patterns in time series, leading us to the SARIMA model.

The **Seasonal Autoregressive Integrated Moving Average model**, or **SARIMA$(p,d,q)(P,D,Q)_m$** adds another set of parameters that allows us to take into account periodic patterns when forecasting a time series which is not always possible with an ARIMA(p,d,q) model.

In this chapter, we examine the SARIMA$(p,d,q)(P,D,Q)_m$ model and adapt the general modeling procedure to account for the new parameters. We also determine how to identify seasonal patterns in a time series, and finally apply the SARIMA model to forecast a seasonal time series.

Specifically, we apply the model to forecast the total number of monthly passengers for an airline. The data was recorded from January 1949 to December 1960. The series is shown in figure 8.1.

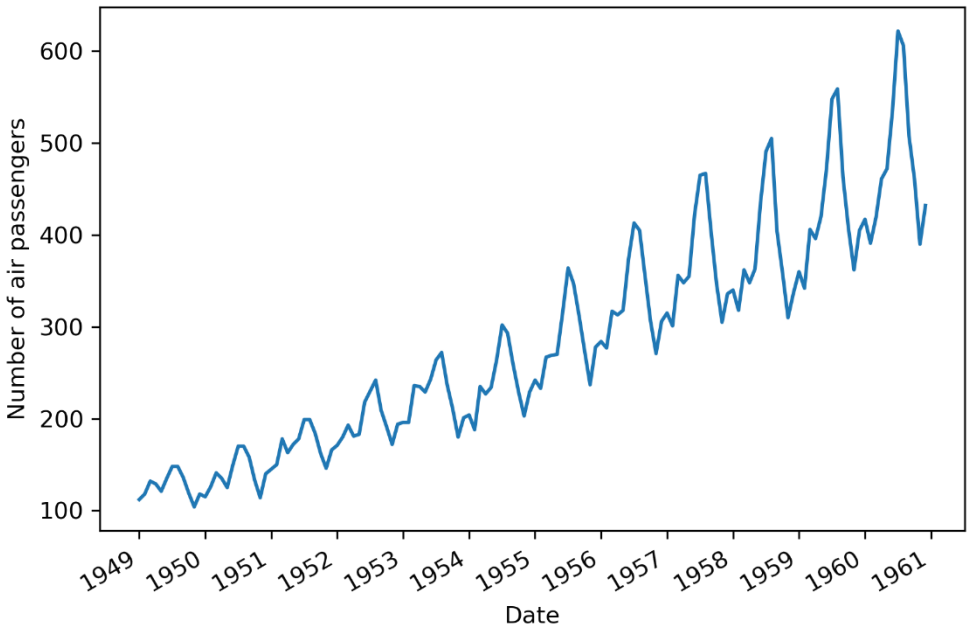


Figure 8.1 Monthly total number of air passengers for an airline, from January 1949 to December 1960. We notice a clear seasonal pattern in the series, with peak traffic occurring towards the middle of each year.

Looking at figure 8.1, we can see a clear seasonal pattern in the series The number of air passengers is lower during the beginning and end of the year, and spikes up during the months of June, July and August. Our objective is then to forecast the number of monthly air passengers for the following year. It is important for an airline company to forecast the number of air passengers, as they can better price their tickets and schedule flights to meet the demand for a given month.

8.1 Examining the SARIMA(p,d,q)(P,D,Q)$_m$ model

The SARIMA(p,d,q)(P,D,Q)$_m$ model further expands the ARIMA(p,d,q) model from the previous chapter by adding seasonal parameters.

We notice the addition of 4 new parameters to the model: *P*, *D*, *Q* and *m*. The first three have the same meaning as in the ARIMA(p,d,q) model, only they are their seasonal counterparts. To understand the meaning of these parameters and how they affect the final model, we must first define the parameter *m*.

The parameter *m* stands for the frequency. In the context of time series, the frequency is defined as the number of observations per cycle.

Now, the length of the cycle will depend on the dataset. For data that was recorded every year, quarter, month, or week, the length of a cycle is considered to be one year. In the case

where the data is recorded annually, then $m = 1$, since there is only one observation per year. If the data is recorded quarterly, then $m = 4$, since there are four quarters in a year, therefore four observations per year. Of course, if the data is recorded monthly, then $m = 12$. Finally, for weekly data, $m = 52$. Table 8.1 indicates the appropriate value of m depending on the frequency at which the data was collected.

Table 8.1 Appropriate frequency m depending on the data

Data	Frequency m
Annual	1
Quarterly	4
Monthly	12
Weekly	52

When data is collected on a daily or sub-daily basis, then there are multiple ways of interpreting the frequency. For example, daily data can have a weekly seasonality. In that case, the frequency is $m = 7$ because there would be 7 observations in a full cycle of one week. It could also have a yearly seasonality, meaning that $m = 365$. Thus, we can see how daily and sub-daily data can have a different cycle length, and therefore a different frequency m. Table 8.2 provides the appropriate value of m depending on the seasonal cycle for daily and sub-daily data.

Table 8.2 Appropriate frequency *m* for daily and sub-daily data

	Frequency m				
Data	Minute	Hour	Day	Week	Year
Daily				7	365
Hourly			24	168	8766
Every minute		60	1440	10080	525960
Every second	60	3600	86400	604800	31557600

Now that we understand the meaning of the parameter m, the meaning of P, D, and Q become intuitive. As mentioned before, they are the seasonal counterparts of the p, d, and q parameters that we know from the ARIMA(p,d,q) model.

Seasonal autoregressive integrated moving average model

The seasonal autoregressive integrated moving average model, or SARIMA, adds seasonal parameters to the ARIMA(p,d,q) model.

It is denoted as SARIMA(p,d,q)(P,D,Q)$_m$, where P is the order of the seasonal AR(P) process, D is the seasonal order of integration, Q is the order of the seasonal MA(Q) process, and m is the frequency or the number of observations per seasonal cycle.

Note that a SARIMA(p,d,q)(0,0,0)$_m$ model is equivalent to an ARIMA(p,d,q) model.

Let's consider an example where $m = 12$. If $P = 2$, this means that we include two past values of the series at lag that is a multiple of m. Therefore, we include the value at y_{t-12} and y_{t-24}.

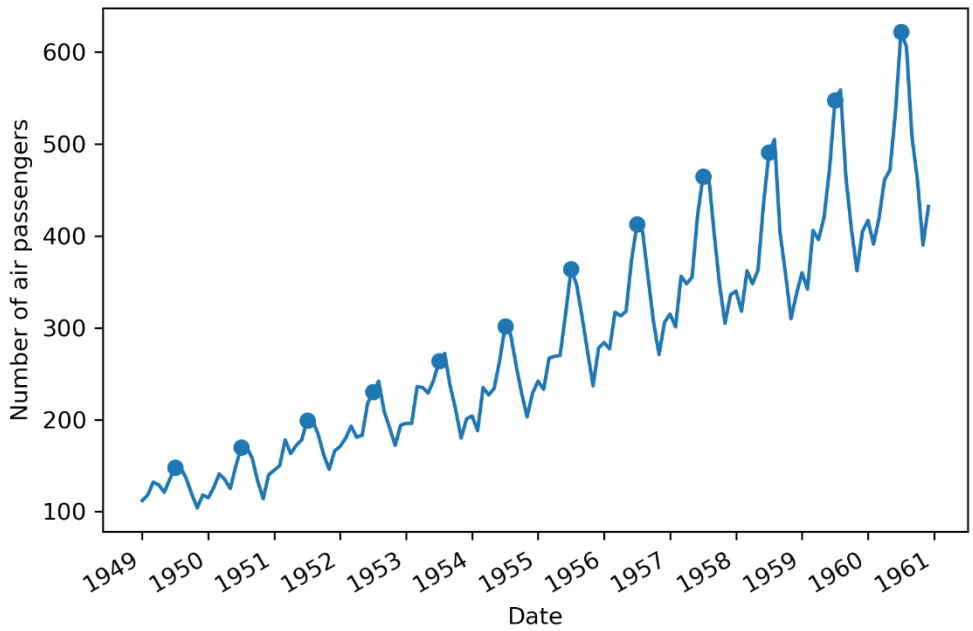
Similarly, if $D = 1$, then it means that a seasonal difference makes the series stationary. In this case, a seasonal difference would be expressed as equation 8.1.

$$y'_t = y_t - y_{t-12}$$

Equation 8.1

Then, in a situation where Q = 2, this means that we include past error terms at lags that are a multiple of m. Therefore, we include the error ε_{t-12} and ε_{t-24}. Let's put this in perspective using the airline's total monthly air passengers dataset.

Knowing that that it is monthly data, it means that $m = 12$. Then, we can see that the month of July and August usually see the highest number of air passengers in the year, as shown by the round markers in figure 8.2. Therefore, if we are to forecast the month of July in 1961, then the information coming from the month of July in prior years is likely going to be useful, since we can intuitively expect the number of air passengers to be at its highest point in the month of July 1961.



the highest number of air passengers in the year. That kind of information is captured by the seasonal parameters *P*, *D*, *Q* and *m* of the SARIMA(p,d,q)(P,D,Q)$_m$ model.

Thus, the parameters *P*, *D*, *Q* and *m* allow us to capture that information present in the previous seasonal cycle to help us forecast our time series.

Now that we have examined the SARIMA model and understand how it expands on the ARIMA model, let's move on to identifying the presence of seasonal patterns in a time series.

8.2 Identifying seasonal patterns in a time series

Intuitively, we know that it makes sense to apply the SARIMA model on data that exhibits a seasonal pattern. Therefore, it is important to determine ways to identify seasonality in time series.

Usually, plotting the time series data is enough to observe periodic patterns. For example, looking at the total monthly air passengers in figure 8.3, it is easy for us to identify a repeating pattern every year, with a high number of passengers being recorded during June, July and August of each year, and less passengers in November, December, and January of each year.

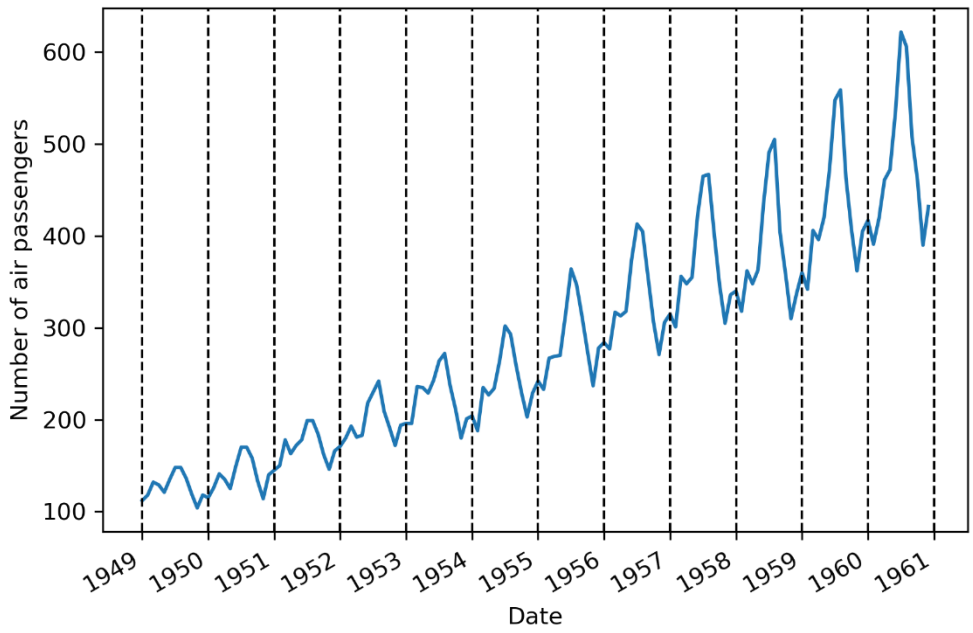


Figure 8.3 Highlighting the seasonal pattern in the monthly number of air passengers. Each dashed vertical line is separated by a period of twelve months. We can clearly see how a peak occurs in the middle of each year, as well as a very similar pattern for the beginning and end of each year. This observation is usually enough to determine that the dataset is seasonal.

Another way of identifying seasonal patterns in a time series is using time series decomposition, a method that we first used in chapter 1. Time series decomposition is a statistical task that separates the time series into its three main components: a trend component, a seasonal component, and the residuals.

The trend component represents the long-term change in the time series. This component is responsible for time series that increase or decrease over time. The seasonal component is of course the seasonal pattern in the time series. It represents repeated fluctuations that occur over a fixed period of time. Finally, the residuals, or the noise, express any irregularity that cannot be explained by the trend or the seasonal component.

Time series decomposition

Time series decomposition is a statistical task that separates the time series into its three main components: a trend component, a seasonal component, and the residuals.

The trend component represents the long-term change in the time series. This component is responsible for time series that increase or decrease over time. The seasonal component is of course the periodic pattern in the time series. It represents repeated fluctuations that occur over a fixed period of time. Finally, the residuals, or the noise, express any irregularity that cannot be explained by the trend or the seasonal component.

With time series decomposition, we can clearly identify and visualize the seasonal component of a time series. Here, we decompose the dataset for air passengers using the `STL` function from the `statsmodels` library to generate figure 8.4.

```
from statsmodels.tsa.seasonal import STL

decomposition = STL(df.Passengers, period=12).fit()    #A

fig, (ax1, ax2, ax3, ax4) = plt.subplots(nrows=4, ncols=1, sharex=True,
       “[CA]”figsize=(10,8))    #B

ax1.plot(decomposition.observed)
ax1.set_ylabel('Observed')

ax2.plot(decomposition.trend)
ax2.set_ylabel('Trend')

ax3.plot(decomposition.seasonal)
ax3.set_ylabel('Seasonal')

ax4.plot(decomposition.resid)
ax4.set_ylabel('Residuals')

plt.xticks(np.arange(0, 145, 12), np.arange(1949, 1962, 1))

fig.autofmt_xdate()
plt.tight_layout()
```

#A Decompose the series using the `STL` function. The period is equal to the frequency `m`. Here, since we have monthly data, the period is then 12.

#B Plot each component in a figure.

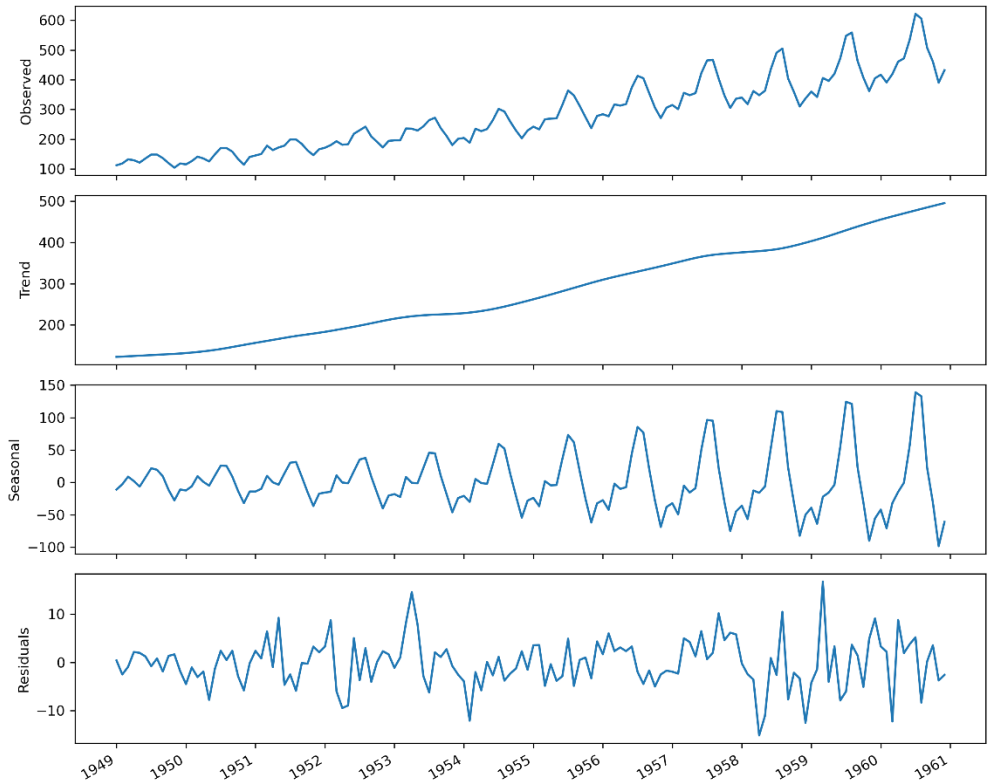


Figure 8.4 Decomposition of the dataset for air passengers. The first plot shows the observed data. The second plot shows the trend component, which tells us that the number of air passengers is increasing over time. The third plot displays the seasonal component, and we can clearly see a repeating pattern through time. Finally, the last plot shows the residuals, which are variations in the data that cannot be explained by the trend or the seasonal component.

Looking at figure 8.4, we can see each component of our time series. The first plot shows the data itself. Then, the second plot shows the trend component, which is increasing over time. The third plot displays the seasonal component, and we can clearly see a repeating pattern over time, which is a strong hint that using the SARIMA model is appropriate in this case. Finally, the last plot shows the residuals, or the noise, which are the random variations in the data that are not explained by the trend and the seasonal component.

We also notice that the y-axis for the plots of the trend, seasonal and residuals component are all slightly different from the observed data. This is because each plot shows the magnitude of change that is attributed to that particular component. That way, the sum of the trend, seasonal component, and residuals results in the observed data shown on the top plot. This explains why the seasonal component is sometimes in the negative values and other times in the positive values, as it creates the peaks and troughs in the observed data.

In a situation where we have a time series with no seasonal pattern, then the decomposition process will display a flat horizontal line at 0 for the seasonal component. To demonstrate that, we simulated a linear time series and decomposed it into its three components using the same method as above. The result in shown in figure 8.5.

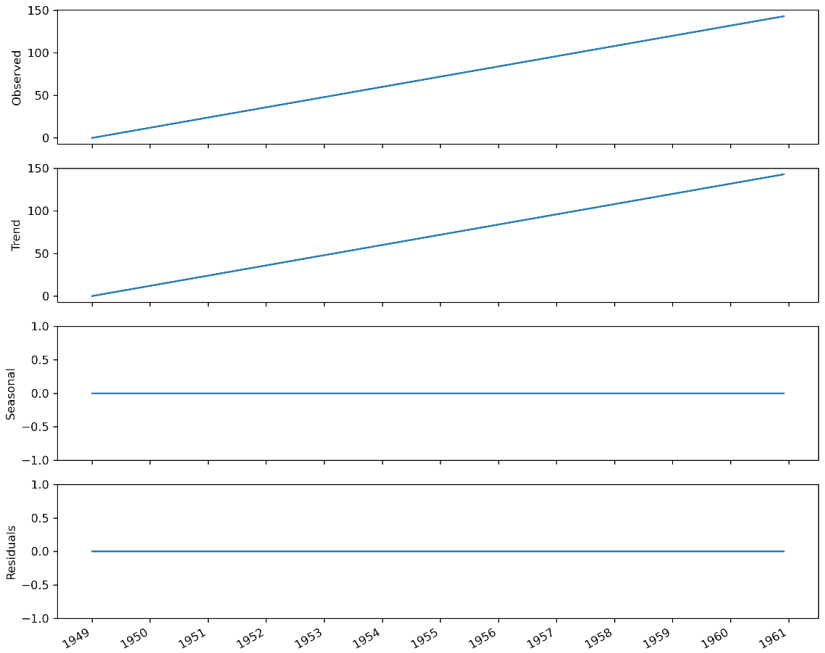


Figure 8.5 Time series decomposition of a simulated linear series. We can see in the top plot the observed data and notice that we simulated a perfectly linear series. The second plot shows the trend component, which is expected to be the same as the observed data, since the series is linearly increasing over time. Since there is no seasonal pattern, the seasonal component is then a flat horizontal line at 0. Here, the residuals are also 0, because we simulated a perfectly linear series.

Looking at the top plot of figure 8.5, we see that we simulated a perfectly linear time series that is increasing over time. The second displays the trend component, and we notice that it is identical to the observed data. This is expected since we simulated a perfectly linear series. Of course, since there is no seasonal pattern in our simulated series, we see that the seasonal component is a flat horizontal line at 0 on the third plot. Finally, the residuals on the fourth plot are also 0, because we simulated a perfectly linear series, so there are no random fluctuations to display.

Therefore, we can see how time series decomposition can help us determine if our data is seasonal or not. While it is a graphical method and not a statistical test, it is enough to determine if a series is seasonal or not, so that we can apply the appropriate model for forecasting. In fact, there are no statistical tests to identify seasonality in time series.

Now that we know how to identify seasonal patterns in our series, we can move on to adapting the general modeling procedure to include the new parameters of the SARIMA$(p,d,q)(P,D,Q)_m$ model and forecast the number of monthly air passengers.

8.3 Forecasting the number of monthly air passengers

In the previous chapter, we adapted our general modeling procedure to account for the new parameter d coming from the ARIMA model that allows us to forecast non-stationary time series. The steps are outlined in figure 8.6. Now, we must modify it again in order to account for the new parameters of the SARIMA model, which are *P*, *D* and *Q* and *m*.

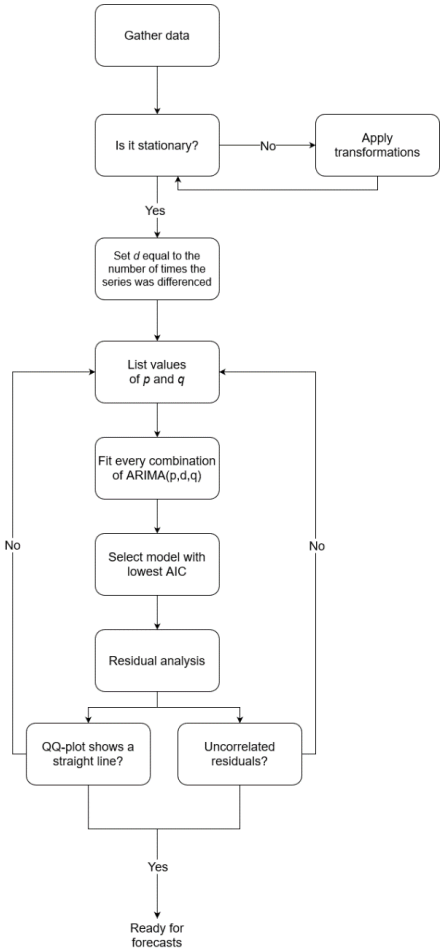


Figure 8.6 General modeling procedure for an ARIMA model. We now need to adapt the steps to account for the parameters *P*, *D*, *Q* and *m* of the SARIMA model.

The first step of gathering data remains untouched. Then, we still check for stationarity and apply transformation in order to set the parameter *d*. However, we can also perform seasonal differencing to make the series stationary. Then *D* will be equal to minimum number of times we applied seasonal differencing to make the series stationary.

Then, we set a range of possible values for *p*, *q*, *P*, and *Q*, as the SARIMA model can also incorporate the order of the seasonal autoregressive process and seasonal moving average process. Note that the addition of two new parameters will increase the number of unique combinations of SARIMA$(p,d,q)(P,D,Q)_m$ models we can fit, so this step will take longer to complete. The rest of the procedure remains the same, as we still select the model with the lowest AIC and perform residual analysis before using the model for forecasting. The resulting modeling procedure is shown in figure 8.7.

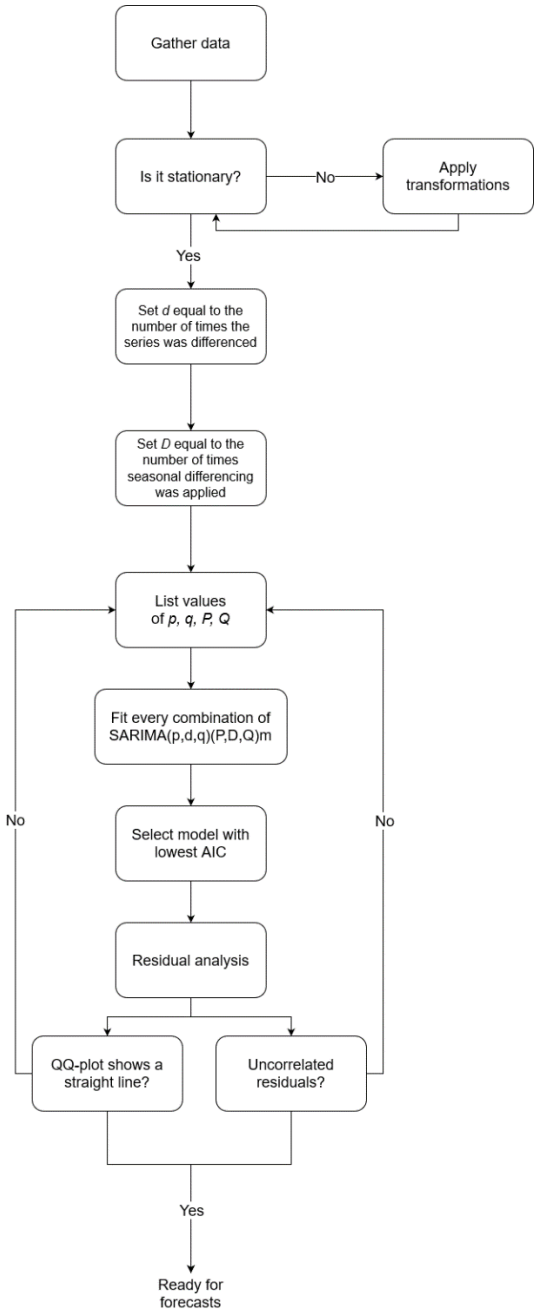


Figure 8.7 General modeling procedure for the SARIMA model. Note that we can set *P*, *D* and *Q* to 0 to obtain an ARIMA(p,d,q) model.

With our new modeling procedure defined, we are now ready to forecast the total number of monthly air passengers. For this scenario, we wish to forecast one year of monthly air passengers. Therefore, we will use the data from the 1960 for the test set as shown in figure 8.8.

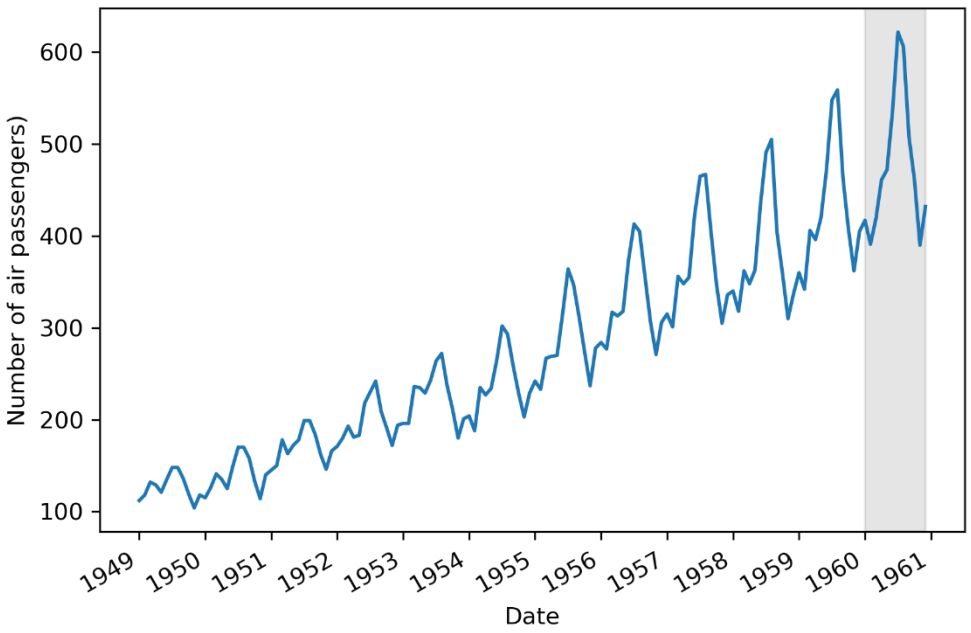


Figure 8.8 Training and test split for the air passengers dataset. The shaded area represents the testing period, which corresponds to the full year of 1960, as our goal is to forecast a year of monthly air passengers.

The baseline model will be the naïve seasonal forecast and we will use both an ARIMA(p,d,q) and a SARIMA(p,d,q)(P,D,Q)$_m$ model to verify if the addition of seasonal components will yield better forecasts.

8.3.1 Forecasting with an ARIMA(p,d,q) model

We first model the dataset using an ARIMA(p,d,q) model. That way, we can compare its performance to the SARIMA(p,d,q)(P,D,Q)$_m$ model.

Following the general modeling procedure we outlined before, we first test for stationarity. Again, we use the ADF test.

```
ad_fuller_result = adfuller(df.Passengers)

print(f'ADF Statistic: {ad_fuller_result[0]}')
print(f'p-value: {ad_fuller_result[1]}')
```

This prints out an ADF statistic of 0.82 and a p-value of 0.99. Therefore, we cannot reject the null hypothesis and the series is not stationary. Hence, we difference the series and test for stationarity again.

```
eps_diff = np.diff(df.Passengers, n=1)    #A

ad_fuller_result = adfuller(eps_diff)

print(f'ADF Statistic: {ad_fuller_result[0]}')
print(f'p-value: {ad_fuller_result[1]}')
```

#A First-order differencing

This returns an ADF statistic of -2.83 and a p-value of 0.054. Again, we cannot reject the null hypothesis, and differencing the series once did not make it stationary. Therefore, we difference it again and test for stationarity.

```
eps_diff2 = np.diff(eps_diff, n=1)    #A

ad_fuller_result = adfuller(eps_diff2)

print(f'ADF Statistic: {ad_fuller_result[0]}')
print(f'p-value: {ad_fuller_result[1]}')
```

#A Series is now differenced twice

This returns an ADF statistic of -16.38 and a p-value of 2.73×10^{-29}. Now, we can reject the null hypothesis and our series is considered to be stationary. Since the series was differenced twice to become stationary, $d = 2$.

Now, we can define a range of possible values for the parameters p and q and fit all unique ARIMA(p,d,q) models. Here, each parameter can be given a value between 0 and 12. We specifically choose this range to allow the ARIMA model to go back 12 timesteps in time. Since the data is sampled monthly and we know it is seasonal, we can hypothesize that the number of air passengers in January of a given year is likely predictive of the number of air passengers in January of the following year. Since these two points are 12 timesteps apart, we allow the values of p and q to vary between 0 and 12 in order to potentially capture this seasonal information in the ARIMA(p,d,q) model. Finally, since we are working with an ARIMA model, we set P, D, and Q to 0. Note the use of the parameter `s` which is equivalent to m. The implementation of SARIMA in `statsmodels` simply uses `s` instead of m, and they both denote the frequency.

```
ps = range(0, 13, 1)    #A
qs = range(0, 13, 1)
Ps = [0]    #B
Qs = [0]

d = 2    #C
D = 0    #D
s = 12    #E

ARIMA_order_list = list(product(ps, qs, Ps, Qs))    #F
```

#A Allow p and *q* to vary between 0 and 12 in order to capture seasonal information.

#B Set *P* and *Q* to 0, since we are working with an ARIMA(p,d,q) model.
#C Set the parameter d to number of times the series was differenced to become stationary
#D *D* is set to 0 since we are working with an ARIMA(p,d,q) model.
#E The parameter `s` is equivalent to *m*. They both denote the frequency. It is simply how the SARIMA model is implemented in the `statsmodels` library.
#F Generate all possible combination of (p,d,q)(0,0,0).

We notice that we set the parameters *P*, *D*, *Q* and *m*, even though we work with an ARIMA model. This is because we define the `optimize_SARIMA` function which will then be reused in the next section. Remember that we set *P*, *D*, *Q* to 0, because a SARIMA(p,d,q)$(0,0,0)_m$ model is equivalent to an ARIMA(p,d,q) model.

The function `optimize_SARIMA` build upon the function `optimize_ARIMA` that we defined in the previous chapter. This time, we integrate the possible values of *P* and *Q*, as well as add the seasonal order of integration D and the frequency *m*. The function is shown in listing 8.1.

Listing 8.1 Defining a function to select the best SARIMA model

```
from typing import Union
from tqdm import tqdm_notebook
from statsmodels.tsa.statespace.sarimax import SARIMAX

def optimize_SARIMA(endog: Union[pd.Series, list], order_list: list, d: "[CA]"int, D: int,
       s: int) -> pd.DataFrame:    #A

    results = []

    for order in tqdm_notebook(order_list):    #B
        try:
            model = SARIMAX(
                endog,
                order=(order[0], d, order[1]),
                seasonal_order=(order[2], D, order[3], s),
                simple_differencing=False).fit(disp=False)
        except:
            continue

        aic = model.aic
        results.append([order, model.aic])

    result_df = pd.DataFrame(results)
    result_df.columns = ['(p,q,P,Q)', 'AIC']

    #Sort in ascending order, lower AIC is better
    result_df = result_df.sort_values(by='AIC',
       "[CA]"ascending=True).reset_index(drop=True)

    return result_df    #C
```

#A The `order_list` parameter now includes *p*, *q*, *P* and *Q* orders. We also add the seasonal order of differencing *D* and the frequency. Remember that the frequency *m* in the SARIMA model is denoted as `s` in the implementation in the `statsmodels` library.
#B Loop over all unique SARIMA(p,d,q)(P,D,Q)$_m$, fit them and store the AIC.
#C Return the sorted `DataFrame`, starting with the lowest AIC.

With the function ready, we can launch it using the training set and get the ARIMA model with the lowest AIC. Despite that fact that we are using the `optimize_SARIMA` function, we are still fitting an ARIMA model, because we specifically set *P*, *D* and *Q* to 0. For the training set, we take all data points but the last twelve, as they will be used for the test set.

```
train = df.Passengers[:-12]    #A

ARIMA_result_df = optimize_SARIMA(train, ARIMA_order_list, d, D, s)    #B
ARIMA_result_df    #C
```

#A The training set consists of all data points but the last 12, as the last year of data is used for the test set.
#B Run the `optimize_SARIMA` function.
#C Display the sorted `DataFrame` in increasing order of AIC.

This returns a `DataFrame` where the model with the lowest AIC is a SARIMA(11,2,3)$(0,0,0)_{12}$ model, which is equivalent to an ARIMA(11,2,3) model. As we can see, allowing the order p to vary between 0 and 12 was beneficial for the model, as the model with the lowest AIC takes into account the past 11 values of the series, since $p = 11$. We will see if this is enough to capture seasonal information from the series, and we will compare the performance of the ARIMA model to the SARIMA model in the next section.

For now, we focus on performing residual analysis. We fit the ARIMA(11,2,3) model obtained previously and plot the residuals' diagnostics. The result is shown in figure 8.9.

```
ARIMA_model = SARIMAX(train, order=(11,2,3), simple_differencing=False)
ARIMA_model_fit = ARIMA_model.fit(disp=False)

ARIMA_model_fit.plot_diagnostics(figsize=(10,8));
```

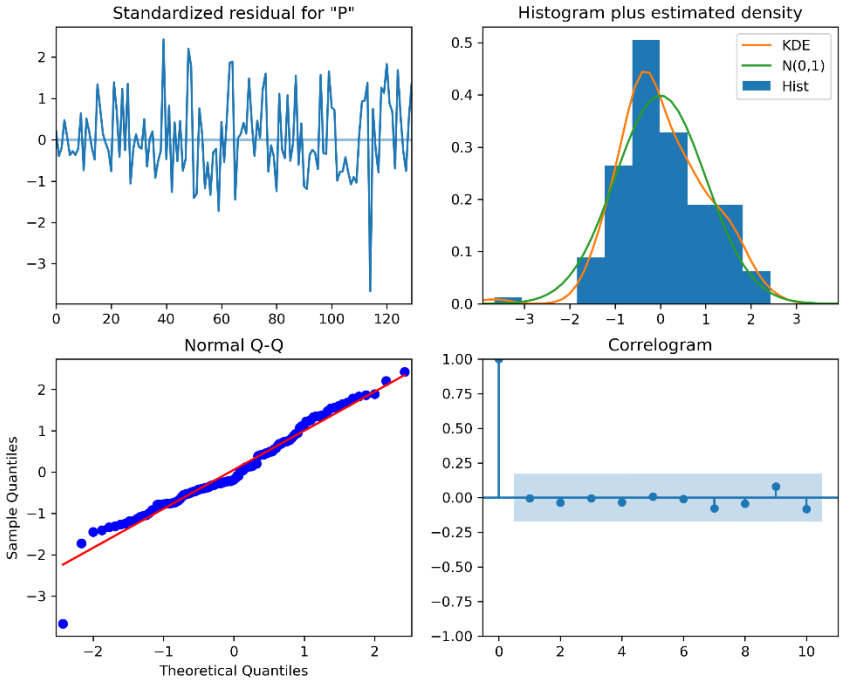


Figure 8.9 Residuals' diagnostics of the ARIMA(11,2,3) model. Looking at the top left plot, we can see that the residuals have no trend with a variance that seems fairly constant over time, which resembles the behavior of white noise. The top right plot shows the distribution of the residuals which approaches a normal distribution. This is further confirmed by the Q-Q plot on the bottom left plot which displays a fairly straight line that lies on $y = x$. Finally, the correlogram on the bottom right plot shows no significant autocorrelation coefficients after lag 0, which is exactly like white noise. From this analysis, the residuals resemble white noise.

Looking at figure 8.9, we start our analysis by studying the top left plot. This displays the standardized residuals over time. We see that there is no trend, and the variance looks constant over time, which is a good point since this behavior is similar to that of white noise. Then, the top right plot shows the distribution of the residuals. Ideally, it should approach a normal distribution. We can see it is fairly close to a normal distribution, despite the unusual peak. The Q-Q plot on the bottom left further supports that, since we see reasonably straight line that lies on $y = x$. Finally, the bottom right plot shows the correlogram of the residuals. We can see that there are not significant autocorrelation coefficients after lag 0, just like white noise. Thus, from the qualitative analysis, the residuals are close to white noise, meaning that the errors are random.

The next step is to run the Ljung-Box test on the residuals to make sure that they are independent and uncorrelated.

```
from statsmodels.stats.diagnostic import acorr_ljungbox

residuals = ARIMA_model_fit.resid

lbvalue, pvalue = acorr_ljungbox(residuals, np.arange(1, 11, 1))

print(pvalue)
```

The returned p-values are all greater than 0.05. Therefore, we do not reject the null hypothesis, meaning that the residuals are indeed uncorrelated, just like white noise. Therefore, the model passes all tests from the residuals analysis, and it can be used for forecasting.

As previously mentioned, we wish to predict a full year of monthly air passengers. Therefore, we use the last 12 months of data as our test set. The baseline model is the naïve seasonal forecast, where we simply use number of air passengers for each month of 1959 as a forecast for each month of 1960.

```
test = df.iloc[-12:]    #A

test['naive_seasonal'] = df['Passengers'].iloc[120:132].values    #B
```

#A Create the test set. It corresponds to the last 12 data points, which is the data for the 1960.
#B The naïve seasonal forecasts simply reuse the data from 1959 as a forecast for 1960.

Then, we can append the forecasts from our ARIMA(11,2,3) model to the `test` DataFrame.

```
ARIMA_pred = ARIMA_model_fit.get_prediction(132, 143).predicted_mean    #A

test['ARIMA_pred'] = ARIMA_pred    #B
```

#A Get predictions for each month of 1960
#B Append predictions to `test`.

With forecasts from the ARIMA model stored in test, we will now use a SARIMA model and later compare the performance to see if the SARIMA model actually performs better than an ARIMA model when applied on a seasonal time series.

8.3.2 Forecasting with a SARIMA(p,d,q)(P,D,Q)$_m$ model

In the previous section, we used an ARIMA(11,2,3) model to forecast the number of monthly air passengers. Now, we fit a SARIMA model and determine if it performs better than the ARIMA model. Hopefully, the SARIMA model will perform better, since it can capture seasonal information, and we know that our dataset exhibits a clear seasonality as shown in figure 8.10.

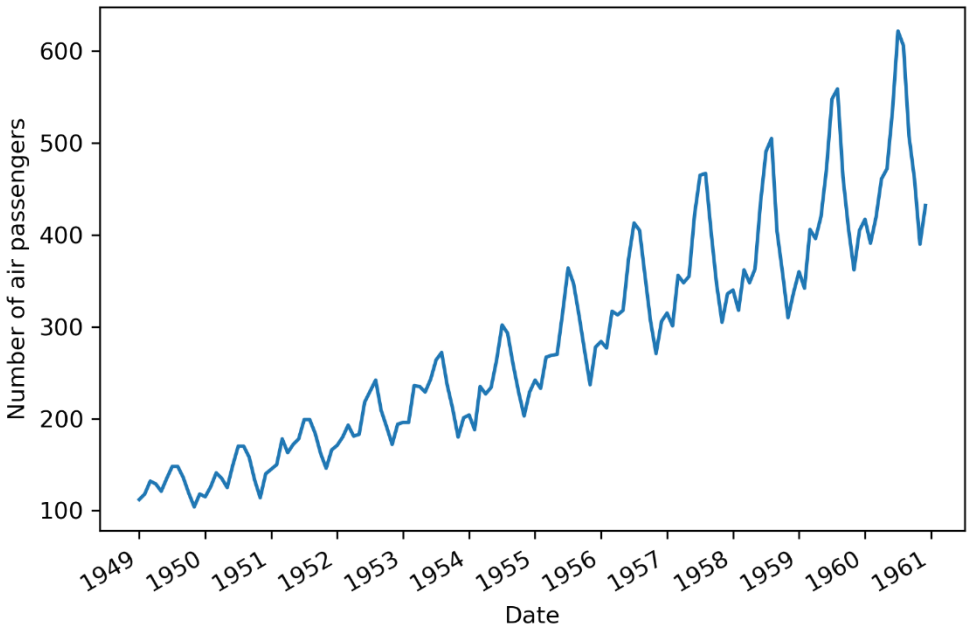


Figure 8.10 Monthly total number of air passengers for an airline, from January 1949 to December 1960. We notice a clear seasonal pattern in the series, with peak traffic occurring towards the middle of the year.

Following the steps in the general modeling procedure in figure 8.11, we first check for stationarity and applied the required transformations.

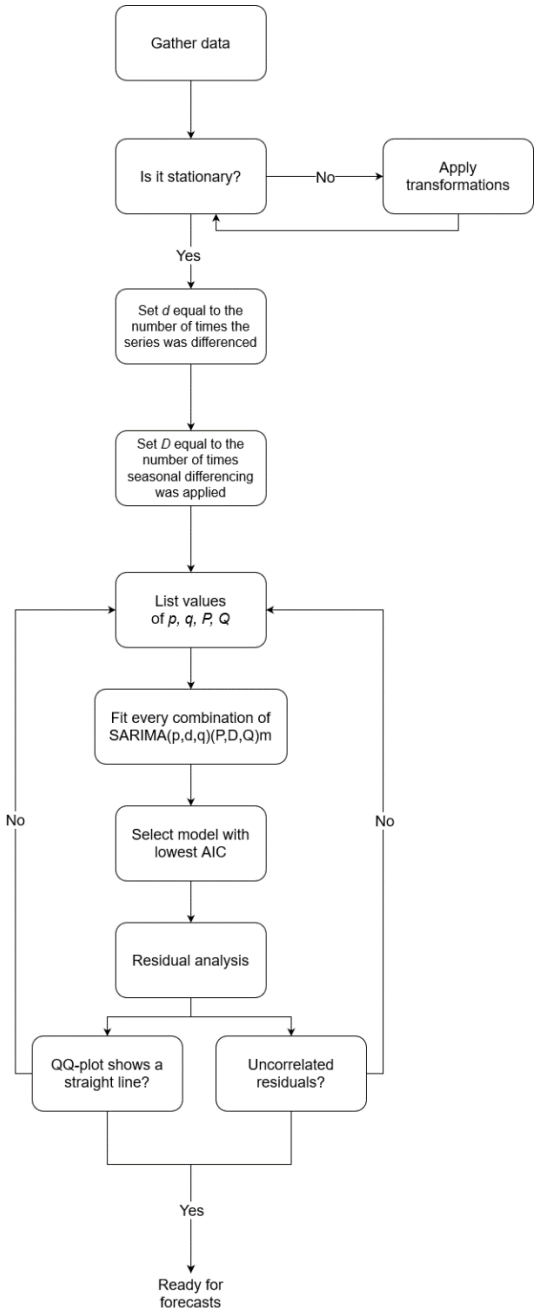


Figure 8.11 General modeling procedure for the SARIMA model. Note that we can set *P*, *D* and *Q* to 0 to obtain an ARIMA(p,d,q) model.

```
ad_fuller_result = adfuller(df.Passengers)

print(f'ADF Statistic: {ad_fuller_result[0]}')
print(f'p-value: {ad_fuller_result[1]}')
```

The ADF test on the dataset itself returns an ADF statistic of 0.82 and a p-value of 0.99. Therefore, we cannot reject the null hypothesis and the series is not stationary. Hence, we apply a first-order differencing and test for stationarity.

```
eps_diff = np.diff(df.Passengers, n=1)

ad_fuller_result = adfuller(eps_diff)

print(f'ADF Statistic: {ad_fuller_result[0]}')
print(f'p-value: {ad_fuller_result[1]}')
```

This returns an ADF statistic of -2.83 and a p-value p-value of 0.054. Since the p-value is greater than 0.05, we cannot reject the null hypothesis and the series is still non-stationary. Therefore, let's apply a seasonal difference and test for stationarity.

```
eps_diff_seasonal_diff = np.diff(eps_diff, n=12)    #A

ad_fuller_result = adfuller(eps_diff_seasonal_diff)

print(f'ADF Statistic: {ad_fuller_result[0]}')
print(f'p-value: {ad_fuller_result[1]}')
```

#A Seasonal differencing. Since we have monthly data, *m = 12*, so the seasonal difference is the difference between two values that are 12 timesteps apart.

This returns an ADF statistic of -17.63 and a p-value of 3.82×10^{-30}. With a large and negative ADF statistic, and a p-value smaller than 0.05, we can reject the null hypothesis and consider the transformed series as stationary. Therefore, we performed one round of differencing, meaning that d = 1, and one round of seasonal differencing, meaning that D = 1.

With this step done, we can now define the range of possible values for p, q, P and Q, fit each unique SARIMA(p,d,q)(P,D,Q)m model, and select the one with the lowest AIC.

```
ps = range(0, 4, 1)    #A
qs = range(0, 4, 1)
Ps = range(0, 4, 1)
Qs = range(0, 4, 1)

SARIMA_order_list = list(product(ps, qs, Ps, Qs))    #B

train = df.Passengers[:-12]    #C

d = 1
D = 1
s = 12

SARIMA_result_df = optimize_SARIMA(train, SARIMA_order_list, d, D, s)    #D
SARIMA_result_df    #F
```

#A We try values of [0,1,2,3] for *p*, *q*, *P* and *Q*.
#B Generate the unique combinations of orders
#C The training set consists of all the data, but the last 12 data points used for the test set.
#D Fit all SARIMA models on the training set.
#E Display the result

Once the function is done running, we obtain that a SARIMA(2,1,1)(1,1,2)$_{12}$ model is the model with the lowest AIC, which has a value of 892.24. We can fit this model again on the training set to perform residual analysis. We start by plotting the residuals' diagnostics in figure 8.12.

```
SARIMA_model = SARIMAX(train, order=(2,1,1), seasonal_order=(1,1,2,12),
       "[CA]"simple_differencing=False)
SARIMA_model_fit = SARIMA_model.fit(disp=False)

SARIMA_model_fit.plot_diagnostics(figsize=(10,8));
```

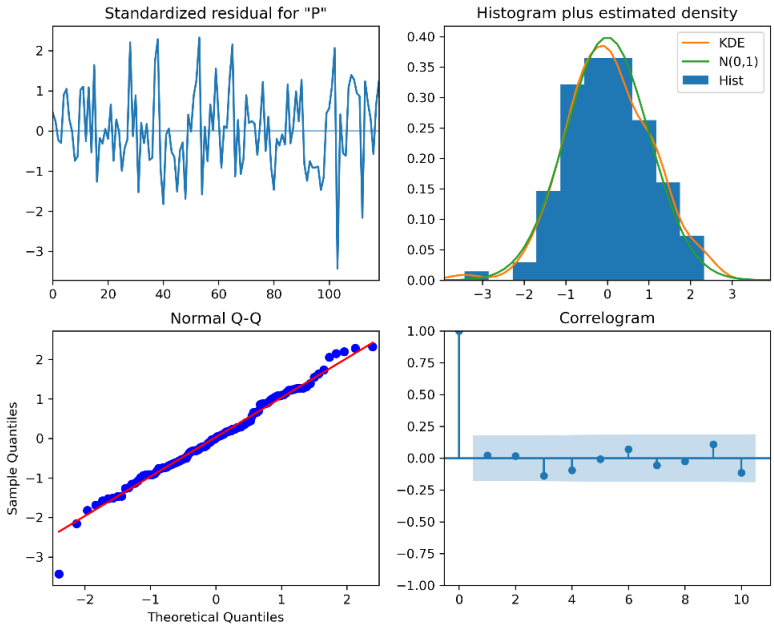


Figure 8.12 Residuals' diagnostics of the SARIMA(2,1,1)(1,1,2)$_{12}$ model. The top left plot shows that the residuals do not exhibit a trend or a change in variance. The top right plot shows that the residuals' distribution is very close to anormal distribution. This is further supported by the Q-Q plot on the bottom left that displays a fairly straight line that lies on $y = x$. Finally, the correlogram on the bottom right shows no significant coefficients after lag 0. Therefore, everything leads to the conclusion that the residuals resemble white noise, which is exactly what we are looking for.

Looking at figure 8.12, we see on the top left plot that the residuals do not exhibit a long-term trend, and variance seems constant over time, just like white noise. On the top right, we can see that the distribution of residuals is very close to the normal distribution. The Q-Q plot on the bottom left further supports that, since we see a fairly straight line that lies on $y = x$. Finally, the correlogram on the bottom right shows no significant coefficients after lag 0, which is expected from random white noise. Therefore, it seems that our residuals are completely random, which is exactly what we are looking for in a good model.

The final test to determine if we can use this model for forecasting or not, is the Ljung-Box test.

```
from statsmodels.stats.diagnostic import acorr_ljungbox

residuals = SARIMA_model_fit.resid

lbvalue, pvalue = acorr_ljungbox(residuals, np.arange(1, 11, 1))

print(pvalue)
```

The returned p-values are all greater than 0.05. Therefore, we do not reject the null hypothesis and conclude that the residuals are independent and uncorrelated, just like white noise.

Our model has passed all tests from the residuals analysis, and we are ready to use it for forecasting. Again, we forecast the number of monthly air passengers for the year of 1960 to compare the predicted values to the observed values in the test set.

```
SARIMA_pred = SARIMA_model_fit.get_prediction(132, 143).predicted_mean  #A

test['SARIMA_pred'] = SARIMA_pred
```

#A Forecast the number of monthly air passengers for the year 1960

Now that we have the observed values, the predicted values from the naïve seasonal forecast, the predicted values from the ARIMA model, and the predicted values from the SARIMA model, we can compare the performance of each and determine the best forecasting method for our problem.

8.3.3 Comparing performance of each forecasting method

We can compare the performance of each forecasting method: the naïve seasonal forecasts, the ARIMA model, and the SARIMA model. We use the mean absolute percentage error or MAPE to evaluate each model.

We can first visualize the forecasts again the observed values of the test set. The plot if shown in figure 8.13.

```
fig, ax = plt.subplots()

ax.plot(df.Month, df.Passengers)
ax.plot(test.Passengers, 'b-', label='actual')
ax.plot(test.naive_seasonal, 'r:', label='naive seasonal')
ax.plot(test.ARIMA_pred, 'k--', label='ARIMA(11,2,3)')
ax.plot(test.SARIMA_pred, 'g-.', label='SARIMA(2,1,1)(1,1,2,12)')
```

```
ax.set_xlabel('Date')
ax.set_ylabel('Number of air passengers')
ax.axvspan(132, 143, color='#808080', alpha=0.2)

ax.legend(loc=2)

plt.xticks(np.arange(0, 145, 12), np.arange(1949, 1962, 1))
ax.set_xlim(120, 143)    #A

fig.autofmt_xdate()
plt.tight_layout()
```

#A Zoom in on the test set

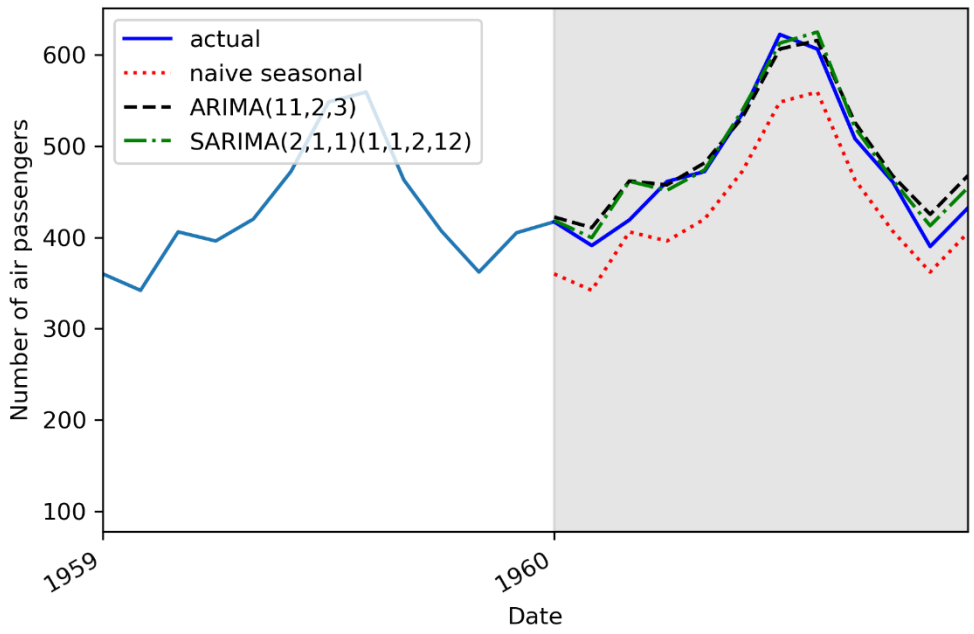


Figure 8.13 Forecasts of the number of monthly air passengers. The shaded area designates the test set. We can see that the curves coming from the ARIMA and SARIMA models almost confound with the observed data, which is indicative of good predictions.

Looking at figure 8.13, we can clearly distinguish the dotted line coming from our baseline from the observed data in the test set. However, both line coming from the ARIMA and SARIMA models sit almost on top of the observed data, meaning that the predictions are very close to the observed data.

We can then measure the MAPE of each model and display it in a bar plot, as shown in figure 8.14.

```
def mape(y_true, y_pred):    #A
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100

mape_naive_seasonal = mape(test.Passengers, test.naive_seasonal)    #B
mape_ARIMA = mape(test.Passengers, test.ARIMA_pred)
mape_SARIMA = mape(test.Passengers, test.SARIMA_pred)

fig, ax = plt.subplots()    #C

x = ['naive seasonal', 'ARIMA(11,2,3)', 'SARIMA(2,1,1)(1,1,2,12)']
y = [mape_naive_seasonal, mape_ARIMA, mape_SARIMA]

ax.bar(x, y, width=0.4)
ax.set_xlabel('Models')
ax.set_ylabel('MAPE (%)')
ax.set_ylim(0, 15)

for index, value in enumerate(y):    #D
    plt.text(x=index, y=value + 1, s=str(round(value,2)), ha='center')

plt.tight_layout()
```

#A Define a function to compute the MAPE
#B Compute the MAPE for each forecasting method
#C Plot the MAPE on a bar plot
#D Display the MAPE as text in the bar plot

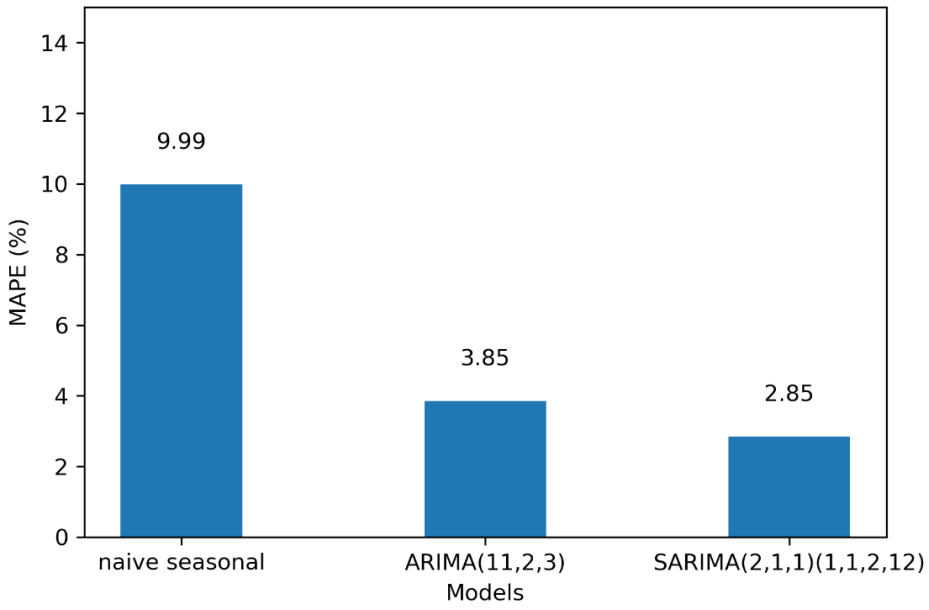


Figure 8.14 MAPE of all forecasting method. We can see that the best performing model is the SARIMA model, since it has the lowest MAPE of all methods.

Looking at figure 8.14, we see that out baseline achieves a MAPE of 9.99%. The ARIMA model produced forecasts with a MAPE of 3.85%, and the SARIMA model scored a MAPE of 2.85%. A MAPE closer to 0 is indicative of better predictions, therefore the SARIMA model is the best performing method for this situation.

This makes sense, since our dataset had a clear seasonality, and the SARIMA model is built to use seasonal properties of time series to make forecasts.

8.4 Next steps

In this chapter, we covered the SARIMA(p,d,q)$(P,D,Q)_m$ model, which allows us to model non-stationary seasonal time series.

The addition of the parameters *P*, *D*, *Q* and *m* allow us to include the seasonal properties of a time series in a model and use them to produce forecasts. Here, *P* is the order of the seasonal autoregressive process, *D* is the order of seasonal integration, *Q* is the order of the seasonal moving average process, and *m* is the frequency of the data.

To apply the SARIMA model, we saw how to first detect seasonal patterns using time series decomposition, and we adapted the general modeling procedure to also test values for *P* and *Q*.

From chapter 4 to 8, we have slowly built a more general and complex model, starting with the MA(q) model, the AR(p) model, combining them into the ARMA(p,q) model, leading us to the ARIMA(p,d,q) model, and finally the SARIMA(p,d,q)$(P,D,Q)_m$ model.

All these models only consider the values of the time series itself. However, it would make sense that external variables are also predictive of our time series. For example, if we wish to model a country's total spending over time, then looking at the interest rates or debt level can likely be predictive. So, how do we include those external variables in a model?

This leads us to the **SARIMAX** model. Notice the addition of X which stands for exogenous variables. This model combines everything that we have learned so far, and further expands on it by adding the effect of external variables to predict our target. This will be the subject of the next chapter.

8.5 Exercises

8.5.1 Medium: Apply the SARIMA(p,d,q)$(P,D,Q)_m$ model on the Johnson & Johnson dataset

In chapter 7, we applied an ARIMA(p,d,q) model to the Johnson & Johnson dataset to forecast the quarterly EPS over a year.

Now, use the SARIMA(p,d,q)$(P,D,Q)_m$ model on the same dataset and compare its performance to the ARIMA model.

- Use time series decomposition to identify the presence of a periodic pattern.
- Use the `optimize_SARIMA` function and select the model with the lowest AIC.
- Perform residual analysis.
- Forecast the EPS for the last year and measure the performance against the ARIMA model. Use the MAPE. Is it better?

8.6 Summary

- The seasonal autoregressive integrated moving average model, denoted as SARIMA(p,d,q)(P,D,Q)$_m$, is the addition of seasonal properties to the ARIMA(p,d,q) model.
- *P* is the order of the seasonal autoregressive process, *D* is the order of seasonal integration, *Q* is the order of the seasonal moving average process, and *m* is the frequency of the data.
- The frequency m corresponds to the number of observations in a cycle. If the data is collected every month, then *m = 12*. If data is collected every quarter, then *m = 4*.
- Time series decomposition can be used to identify seasonal patterns in a time series.

9

Adding external variables to our model

This chapter covers

- Examining the SARIMAX model
- Exploring the caveat of using external variables for forecasting
- Forecasting using the SARIMAX model

From chapter 4 to 8, we have increasingly built a more general model that allows us to consider more complex patterns in time series. We started our journey with only autoregressive and moving average processes, before combining them into the ARMA model. Then, we added a layer of complexity that allows modeling non-stationary time series, leading us to the ARIMA model. Finally, in the last chapter, we added yet another layer to ARIMA, which allows us to consider seasonal patterns in our forecasts, hence reaching the SARIMA model.

Up until now, each model that we have explored and used to produce forecasts considers only the time series itself. In other words, past values of the time series were used as predictors for future values. However, it is possible that external variables also have an impact on our time series and can therefore be good predictors of future values.

This brings us to the **SARIMAX** model. We notice the addition of the **X** term, which denotes exogenous variables. Note that in statistics, the term exogenous is used to describe predictors or input variables, while endogenous is used to define the target variable; what we are trying to predict.

With the SARIMAX model, we can now consider external variables, or exogenous variables, to forecast a time series.

As a guiding example, we use a macroeconomics dataset from the United States, collected quarterly from 1959 to 2009, to forecast the real gross domestic product, or real GDP, as shown in figure 9.1.

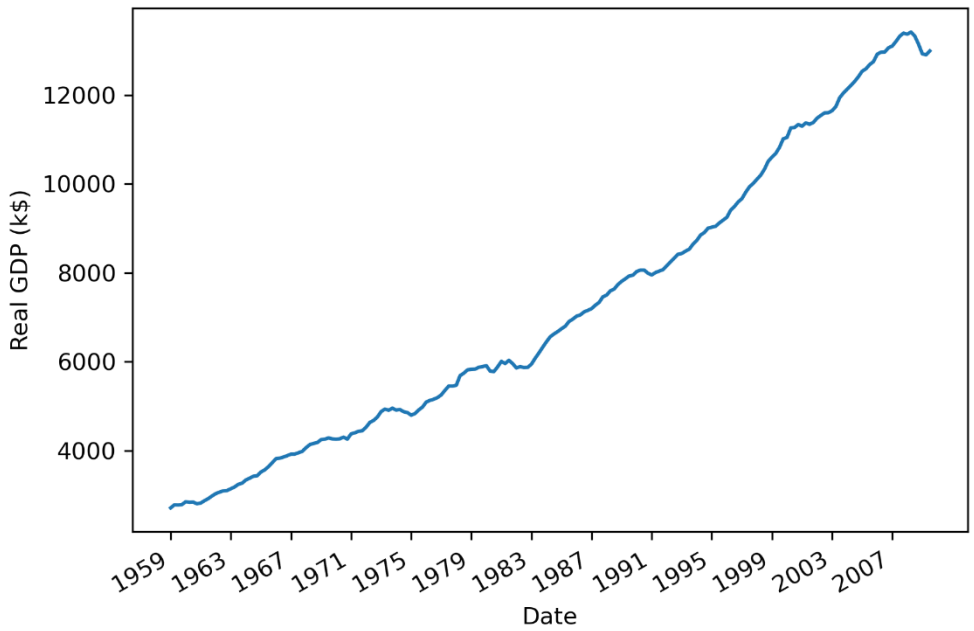


Figure 9.1 Real gross domestic product (GDP) of the United States between 1959 and 2009. The data is collected quarterly and is expressed in thousands of US dollars. Notice the clear positive trend over the years with no cyclical pattern, suggesting that seasonality is not present in the series.

The GDP is the total market value of all the finished goods and services produced within a country. The real GDP is then an inflation-adjusted measure that removes the impact of inflation on the market value of goods. Inflation or deflation can respectively increase or decrease the monetary value of goods and services, hence increasing or decreasing the GDP. By removing the effect of inflation, we can better capture if an economy saw an expansion of production.

Without diving into the technicalities of measuring the GDP, we define the GDP as the sum of consumption C, government spending G, investments I and net exports NX, as shown in equation 9.1.

$$GDP = C + G + I + NX$$

Equation 9.1

Each element of equation 9.1 is likely affected by some external variable. For example, consumption is likely impacted by the unemployment rate, since if less people are employed, consumption is likely to decrease. Interest rates can also have an impact, since if they go up, then it is harder to borrow money, and so spending decreases as well. We can also think of currency exchange rates as having an impact on net exports. A weaker local currency will generally stimulate exports and make imports more expensive. Thus, we see how many exogenous variables can likely impact the real GDP of the United States.

In this chapter, we first examine the SARIMAX model and explore an important caveat when using it to produce forecasts. Then, we apply the model to forecast the real GDP of the United States.

9.1 Examining the SARIMAX model

The SARIMAX model further extends the SARIMA(p,d,q)(P,D,Q)$_m$ model by adding the effect of exogenous variables.

Therefore, we can express the present value y_t simply as a SARIMA(p,d,q)(P,D,Q)m model to which we add any number of exogenous variables X_t as shown in equation 9.2.

$$y_t \rightarrow SARIMA(p,d,q)(P,D,Q)_m + \sum_{i=1}^{n} \beta_i X_t^i$$

Equation 9.2.

From equation 9.2, we know that the SARIMA model is a linear model, as it is a linear combination of past values of the series and error terms. Then, we add another linear combination of different exogenous variables, resulting in SARIMAX being a linear model as well. Note that in SARIMAX, the exogenous variables are continuous and not categorical. This makes sense since a linear regression is performed on the exogenous variables, and a linear regression cannot be applied on categorical variables.

We also notice that we have been using the `SARIMAX` function from `statsmodels` since chapter 4 to implement different models. This is because SARIMAX is the most general function to forecast a time series.

We now understand how a SARIMAX model without exogenous variables is a SARIMA model. We see that a model with no seasonality, but exogenous variables can be denoted as an ARIMAX model. Similarly, a model with no seasonality and no exogenous variables becomes an ARIMA model. Of course, depending on the problematic, different combinations of each portion of the general SARIMAX model will be used.

SARIMAX model

The SARIMAX model simply adds a linear combination of exogenous variables to the SARIMA model. This allows us to model the impact of external variables on the future value of a time series. We can loosely define the SARIMAX model as:

$$y_t = SARIMA(p,d,q)(P,D,Q)_m + \sum_{i=1}^{n} \beta_i X_t^i$$

The SARIMAX model is the most general model for forecasting time series. We see that if we have no seasonal patterns, then it becomes an ARIMAX model. With no exogenous variables, it is a SARIMA model. With no seasonality or exogenous variables, it becomes an ARIMA model.

Theoretically, this sums up the SARIMAX model. Chapters 4 to 8 were purposely ordered in such a way that we incrementally developed the SARIMAX model, making the addition of exogenous variables easy to understand. To reinforce our learning, let's explore the exogenous variables of our dataset.

9.1.1 Exploring the exogenous variables of the US macroeconomics dataset

Let's load the US macroeconomics dataset and explore the different exogenous variables available to us to forecast the real GDP.

This dataset is available with the `statsmodels` library, meaning that we do not need to download and read an external file. We can load the dataset using the `datasets` module of `statsmodels`.

```
import statsmodels.api as sm

macro_econ_data = sm.datasets.macrodata.load_pandas().data    #A
macro_econ_data    #B
```

#A Load the US macroeconomics dataset
#B Display the `DataFrame`

This displays the entire `DataFrame` containing the US macroeconomics dataset. Table 9.1 describes the meaning of each variable.

In table 9.1, we have our target variable, or endogenous variable, which is the real GDP. Then, we have 11 exogenous variables that can be used for forecasting. We see the presence of personal and federal consumption expenditures, interest rate, inflation rate, population, and others.

Table 9.1 Description of all variables in the US macroeconomics dataset

Variable	Description
realgdp	Real gross domestic product (the target variable or endogenous variable)
realcons	Real personal consumption expenditure
realinv	Real gross private domestic investment
realgovt	Real federal consumption expenditure and investment
realdpi	Real private disposable income
cpi	Consumer price index for the end of the quarter
m1	M1 nominal money stock
tbilrate	Quarterly monthly average of the monthly 3-month treasury bill
unemp	Unemployment rate
pop	Total population at the end of the quarter
infl	Inflation rate
realint	Real interest rate

Of course, each variable may or may not be a good predictor of the real GDP. We do not have to perform feature selection, because the linear model will attribute a coefficient close to 0 to exogenous variables that are not significant in predicting the target.

For the sake of simplicity and clarity, we will only work with 6 variables in this chapter: the real GDP, which is out target, and `realcons` to `cpi` in table 9.1 as our exogenous variables.

We can visualize how each variable behaves through time to see if we discern any distinctive patterns. The result is shown in figure 9.2.

```
fig, axes = plt.subplots(nrows=3, ncols=2, dpi=300, figsize=(11,6))

for i, ax in enumerate(axes.flatten()[:6]):    #A
    data = macro_econ_data[macro_econ_data.columns[i+2]]    #B

    ax.plot(data, color='black', linewidth=1)
    ax.set_title(macro_econ_data.columns[i+2])    #C
    ax.xaxis.set_ticks_position('none')
    ax.yaxis.set_ticks_position('none')
    ax.spines['top'].set_alpha(0)
    ax.tick_params(labelsize=6)

plt.setp(axes, xticks=np.arange(0, 208, 8), xticklabels=np.arange(1959, 2010, 2))
fig.autofmt_xdate()
plt.tight_layout()
```

#A Iterate for 6 variables.
#B Skip the `year` and `quarter` columns. That way, we start at `realgdp`.
#C Display the variable's name at the top of the plot.

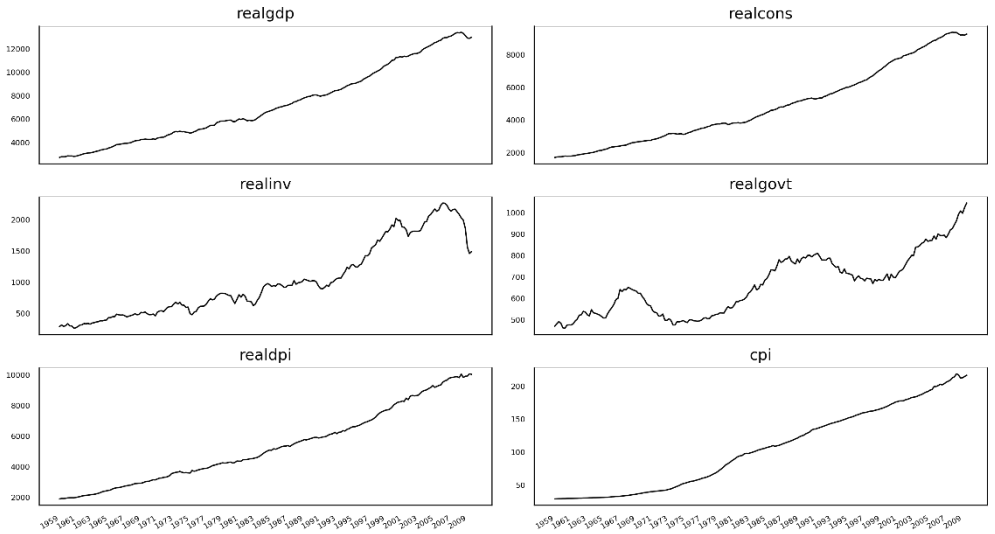


Figure 9.2 Evolution of the real GDP and five exogenous variables between 1959 and 2009. We notice that realgdp, realcons, realdpi and cpi all have a similar shape, which means that realcons, realdpi and cpi are potentially good predictors. On the other hand, realgovt has peaks and troughs that do not appear in realgdp, so we can hypothesize that realgovt is a weaker predictor.

Looking at figure 9.2, we see how realcons, realdpi and cpi all have a similar shape to our target realgdp. This can be a sign that they are good predictors, although a graphical analysis is not sufficient to confirm that idea. On the other hand, realgovt has peaks and troughs that do not appear in realgdp, which may lead us to think that it is not a significant predictor.

Again, the SARIMAX model will automatically find significant and non-significant predictors from the list of exogenous variables. The `statsmodels` library conveniently includes the `summary` method that displays various statistical measures and p-values for each predictor. This method will be used later in the chapter when we forecast the real GDP. If the p-value of a predictor is less than 0.05, then it is correlated with our target. Otherwise, if the p-value is greater than 0.05, it is not correlated.

We should not remove a predictor with a p-value larger than 0.05, because fitting a model without that predictor will change the coefficients and associated p-values. In all cases, we should keep the predictors. If a predictor is not correlated with our target, it will usually have a coefficient close to 0, which basically cancels its effect on predicting the target. We will see an example of that situation when we forecast the real GDP.

9.1.2 Caveat of using SARIMAX

There is an important caveat that comes with the use of the SARIMAX model. Including external variables can potentially be beneficial, as we may find strong predictors for our

target. However, we might encounter issues when forecasting multiple timesteps into the future.

Recall that the SARIMAX model uses the SARIMA(p,d,q)(P,D,Q)$_m$ model and a linear combination of exogenous variable to predict one timestep into the future.

Now, what if we wish to predict two timesteps into the future? While this is possible with a SARIMA model, the SARIMAX model requires us to forecast the exogenous variables too.

To illustrate this idea, let's assume that realcons is a predictor of realgdp (this will be verified later in the chapter). Assume also that we have a SARIMAX model where realcons is used as an input feature to predict realgdp. Now, suppose that we are at the end of 2009 and must predict the real GDP for 2010 and 2011. The SARIMAX model allows us to use the realcons of 2009 to predict the real GDP for 2010. However, predicting the real GDP for 2011 will require us to predict realcons for 2010, unless we wait to observe the value at the end of 2010.

The variable realcons being a time series itself can be forecast using a version of the SARIMA model. Nevertheless, we know that our forecast always has some error associated to it. Therefore, having to forecast an exogenous variable to forecast our target variable can magnify the prediction error of our target, meaning that our predictions can quickly degrade as we predict more timesteps into the future.

The only way to avoid that situation is to predict only one timestep into the future and wait to observe the exogenous variable before predicting the target for another timestep into the future.

On the other hand, if your exogenous variable is *easy* to predict, meaning that it follows a known function that can be accurately predicted, then there is no harm in forecasting the exogenous variable and use these forecasts to predict the target.

In the end, there is no clear recommendation to predict only one timestep. It is dependent on the situation and the exogenous variables available. This is where your expertise as a data scientist and rigorous experimenting comes into play. If you determine that your exogenous variable can be accurately predicted, then you can recommend forecasting many timesteps into the future. Otherwise, your recommendation must be to predict one timestep at a time and justify your decision by explaining that errors will accumulate as more predictions are made, meaning that the forecasts lose accuracy.

Now that we have explored the SARIMAX model in depth, let's apply it to forecast the real GDP.

9.2 Forecasting the real GDP using the SARIMAX model

We are now ready to use the SARIMAX model to forecast the real GDP. Having explored the exogenous variables of the dataset, we will incorporate them into our forecasting model using SARIMAX.

Before diving in, we must reintroduce the general modeling procedure. Here, there are no major changes to the procedure. The only modification is that we now fit a SARIMAX model. All other steps remain the same, as shown in figure 9.3.

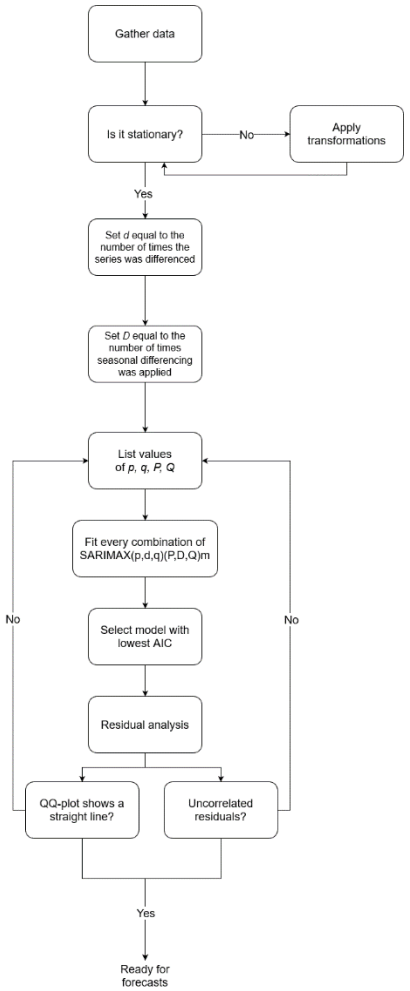


Figure 9.3 General modeling procedure for the SARIMAX model. This procedure can be applied to any problems, as the SARIMAX model is the most general forecasting model and can accommodate all different processes and properties of time series that we have explored. Notice that the only change here is in fitting a SARIMAX model instead of a SARIMA model as shown in chapter 8. The rest of the procedure remains the same.

Following the modeling procedure of figure 9.3, we first check for stationarity of our target using the augmented Dickey-Fuller test of ADF test.

```
target = macro_econ_data['realgdp']    #A
exog = macro_econ_data[['realcons', 'realinv', 'realgovt', 'realdpi', "[CA]"'cpi']]    #B

ad_fuller_result = adfuller(target)
```

```
print(f'ADF Statistic: {ad_fuller_result[0]}')
print(f'p-value: {ad_fuller_result[1]}')
```

#A Define the target variable. In this case, it is the real GDP.
#B Define the exogenous variables. Here, we limit it to 5 variables for simplicity

This returns an ADF statistic of 1.75 and a p-value of 1.00. Since the ADF statistic is not a large negative number, and the p-value is larger than 0.05, we cannot reject the null hypothesis and conclude that the series is not stationary.

Therefore, we must apply a transformation and test for stationarity again. Here, we will difference the series once.

```
target_diff = target.diff()    #A

ad_fuller_result = adfuller(target_diff[1:])

print(f'ADF Statistic: {ad_fuller_result[0]}')
print(f'p-value: {ad_fuller_result[1]}')
```

#A Difference the series

This now returns an ADF statistic of -6.31 and p-value of 3.32×10^{-8}. With a large negative ADF statistic and a p-value smaller than 0.05, we can reject the null hypothesis and conclude that the series is now stationary. Therefore, we know that $d = 1$. Since we did not need to take a seasonal difference the make the series stationary, $D = 0$.

We then define the `optimize_SARIMAX` function, in listing 9.1, that will fit all unique combinations of the model and return a `DataFrame` in ascending order of AIC.

Listing 9.1 Function to fit all unique SARIMAX models

```
from typing import Union
from tqdm import tqdm_notebook
from statsmodels.tsa.statespace.sarimax import SARIMAX

def optimize_SARIMAX(endog: Union[pd.Series, list], exog: Union[pd.Series, "[CA]"list],
       order_list: list, d: int, D: int, s: int) -> pd.DataFrame:

    results = []

    for order in tqdm_notebook(order_list):
        try:
            model = SARIMAX(
                endog,
                exog,    #A
                order=(order[0], d, order[1]),
                seasonal_order=(order[2], D, order[3], s),
                simple_differencing=False).fit(disp=False)
        except:
            continue

        aic = model.aic
        results.append([order, model.aic])

    result_df = pd.DataFrame(results)
    result_df.columns = ['(p,q,P,Q)', 'AIC']
```

```
    #Sort in ascending order, lower AIC is better
    result_df = result_df.sort_values(by='AIC',
       "[CA]"ascending=True).reset_index(drop=True)

    return result_df
```

#A Notice the addition of the exogenous variables when fitting the model.

Then, we define the range of possible values for the orders *p*, *q*, *P* and *Q*. Here, we try values between 0 and 3, but feel free to try a different set of values. Also, since the data is collected quarterly, *m = 4*.

```
p = range(0, 4, 1)
d = 1
q = range(0, 4, 1)
P = range(0, 4, 1)
D = 0
Q = range(0, 4, 1)
s = 4    #A

parameters = product(p, q, P, Q)
parameters_list = list(parameters)
```

#A Remember that `s` in the implementation of SARIMAX from `statsmodels` is equivalent to *m*.

To train the model, we will use the first 200 instances of both the target and exogenous variables. We then run the `optimize_SARIMAX` function and select the model with the lowest AIC.

```
target_train = target[:200]
exog_train = exog[:200]

result_df = optimize_SARIMAX(target_train, exog_train, parameters_list, d, "[CA]"D, s)
result_df
```

Once completed, the function returns that the SARIMAX(3,1,3)$(0,0,0)_4$ model is the model with the lowest AIC. Notice that the seasonal component of the model has only orders of 0. This makes sense, as there is no visible seasonal pattern in the plot of real GDP, as shown in figure 9.4. Therefore, the seasonal component is null, and we therefore have an ARIMAX(3,1,3) model.

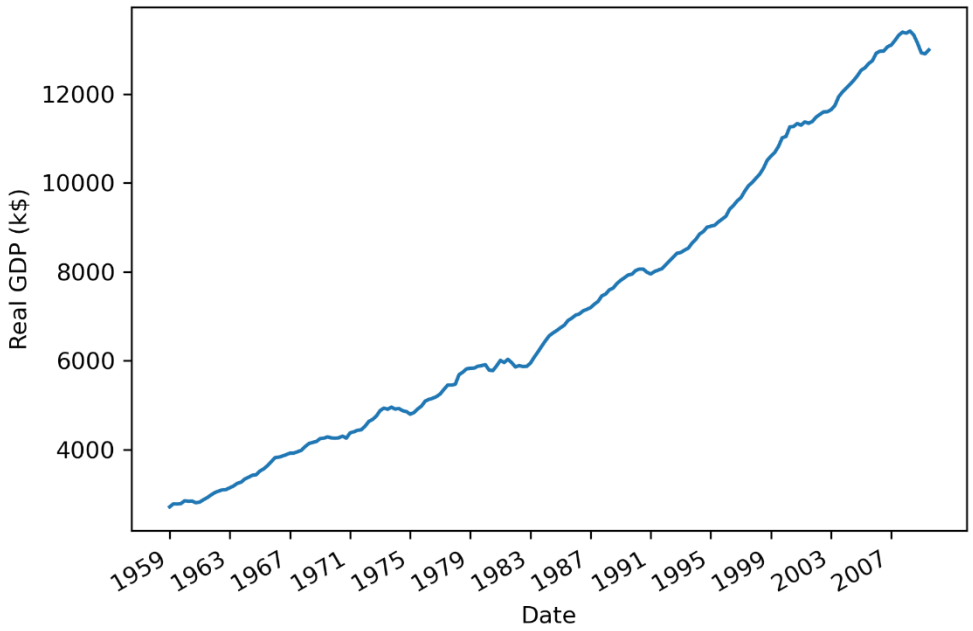


Figure 9.4 Real gross domestic product (GDP) of the United States between 1959 and 2009. The data is collected quarterly and is expressed in thousands of US dollars. Notice the clear positive trend over the years with no cyclical pattern, suggesting that seasonality is not present in the series.

Now, we fit the selected model and display a summary table to see the coefficients associated to our exogenous variables. The result is shown in figure 9.4.

```
best_model = SARIMAX(target_train, exog_train, order=(3,1,3),
       "[CA]"seasonal_order=(0,0,0,4), simple_differencing=False)
best_model_fit = best_model.fit(disp=False)

print(best_model_fit.summary())    #A
```

$A Display the summary table of the model

```
                               SARIMAX Results
==========================================================================================
Dep. Variable:                    realgdp   No. Observations:                  200
Model:                 SARIMAX(3, 1, 3)   Log Likelihood                -859.431
Date:                  Fri, 06 Aug 2021   AIC                           1742.863
Time:                          17:02:59   BIC                           1782.382
Sample:                               0   HQIC                          1758.857
                                  - 200
Covariance Type:                    opg
==========================================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
realcons       0.9652      0.044     21.693      0.000       0.878       1.052
realinv        1.0142      0.033     30.944      0.000       0.950       1.078
realgovt       0.7249      0.127      5.717      0.000       0.476       0.973
realdpi        0.0091      0.025      0.369      0.712      -0.039       0.058
cpi            5.8671      1.311      4.476      0.000       3.298       8.436
ar.L1          1.0648      0.399      2.671      0.008       0.283       1.846
ar.L2          0.4895      0.701      0.698      0.485      -0.885       1.864
ar.L3         -0.6718      0.337     -1.995      0.046      -1.332      -0.012
ma.L1         -1.1035      0.430     -2.565      0.010      -1.947      -0.260
ma.L2         -0.3196      0.767     -0.417      0.677      -1.823       1.184
ma.L3          0.6457      0.403      1.601      0.109      -0.145       1.436
sigma2       328.9706     30.395     10.823      0.000     269.397     388.545
==========================================================================================
Ljung-Box (L1) (Q):                   0.00   Jarque-Bera (JB):                13.55
Prob(Q):                              0.95   Prob(JB):                         0.00
Heteroskedasticity (H):               3.57   Skew:                             0.32
Prob(H) (two-sided):                  0.00   Kurtosis:                         4.11
==========================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

Figure 9.5 Summary table of the selected model. We can see that our exogenous variables were assigned coefficients. We can also see their p-value under the column `P>|z|`. Note that all p-values for our exogenous variables are less than 0.05, but that of `realdpi`. This means that this variable is not significant in predicting our target. Still, we can keep it inside the model, since its coefficient is 0.0091, meaning that its contribution is close to null.

Looking at figure 9.4, we notice that all exogenous variables have a p-value smaller than 0.05, except for `realdpi`, which has a p-value of 0.712. This means that `realdpi` is not correlated with our target `realgdp`. We also notice that its coefficient is 0.0091. Therefore, this variable has almost no impact in the prediction of the target variable.

Moving on with the modeling procedure, we now study residuals of the model shown in figure 9.5.

```
best_model_fit.plot_diagnostics(figsize=(10,8));
```

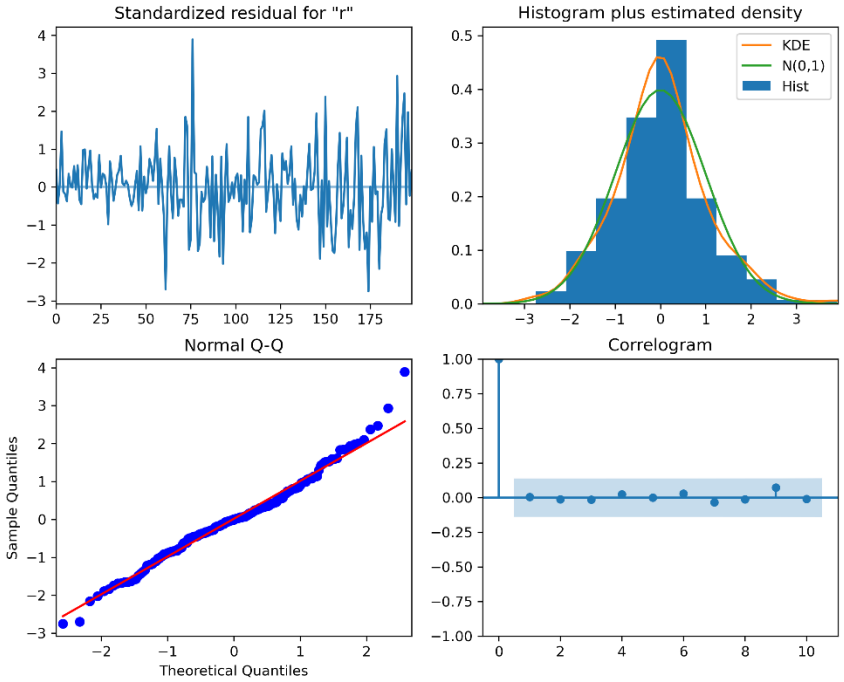


Figure 9.6 Residual analysis of the selected model. We can see that the residuals have no trend and a fairly constant variance over time, just like white noise. On the top right plot, the distribution of residuals is very close to a normal distribution. This is further supported by the Q-Q plot on the bottom left that shows a fairly straight line that lies on $y = x$. Finally, the correlogram shows no significant coefficients after lag 0, just like white noise. Therefore, from a graphical analysis, the residuals of this model resemble white noise.

Looking at figure 9.5, the top left plot shows that the residuals have no trend and a constant variance over time, just like white noise. The top right plot shows that their distribution is very similar to a normal distribution. This is further supported by the Q-Q plot on the bottom left that displays a fairly straight line that lies on $y = x$. Finally, the correlogram on the bottom right shows no significant coefficients after lag 0. Therefore, everything points in the direction that the residuals are completely random, just like white noise. Our model passes the visual check.

Now, we apply the Ljung-Box test to determine if the residuals are not correlated. We therefore want to see p-values that are greater than 0.05, since the null hypothesis of the Ljung-Box test is that residuals are independent and uncorrelated.

```
residuals = best_model_fit.resid
```

```
lbvalue, pvalue = acorr_ljungbox(residuals, np.arange(1, 11, 1))

print(pvalue)
```

All p-values printed are greater than 0.05. Therefore, we do not reject the null hypothesis and conclude that the residuals are independent and uncorrelated. Having passed both residuals checks, our model can be used for forecasting.

As mentioned before, the caveat of using a SARIMAX model is that it is reasonable to predict the next timestep only to avoid predicting the exogenous variables as well, which would lead us to accumulate prediction errors in the final forecast.

Instead, to test our model, we predict the next timestep multiple times and average the error of each prediction. This is done using the `recursive_forecast` function, shown in listing 9.2, that we have defined and worked with in previous chapter. As a baseline model, we will use the last value method.

Listing 9.2 Function to forecast the next timestep multiple times

```
def recursive_forecast(endog: Union[pd.Series, list], exog: “[CA]”Union[pd.Series, list],
       train_len: int, horizon: int, window: int, method: str) -> list:

    total_len = train_len + horizon

    if method == 'last':
        pred_last_value = []

        for i in range(train_len, total_len, window):
            last_value = endog[:i].iloc[-1]
            pred_last_value.extend(last_value for _ in range(window))

        return pred_last_value

    elif method == 'SARIMAX':
        pred_SARIMAX = []

        for i in range(train_len, total_len, window):
            model = SARIMAX(endog[:i], exog[:i], order=(3,1,3),
       “[CA]”seasonal_order=(0,0,0,4), simple_differencing=False)
            res = model.fit(disp=False)
            predictions = res.get_prediction(exog=exog)
            oos_pred = predictions.predicted_mean.iloc[-window:]
            pred_SARIMAX.extend(oos_pred)

        return pred_SARIMAX
```

The `recursive_forecast` function allows us to predict the next timestep over a certain period of time. Specifically, we will use it to forecast the next timestep starting in 2008 to the third quarter of 2009.

```
target_train = target[:196]    #A
target_test = target[196:]    #B

pred_df = pd.DataFrame({'actual': target_test})

TRAIN_LEN = len(target_train)
```

```
HORIZON = len(target_test)
WINDOW = 1    #C

pred_last_value = recursive_forecast(target, exog, TRAIN_LEN, HORIZON, “[CA]”WINDOW,
       'last')
pred_SARIMAX = recursive_forecast(target, exog, TRAIN_LEN, HORIZON, WINDOW,
       “[CA]”'SARIMAX')

pred_df['pred_last_value'] = pred_last_value
pred_df['pred_SARIMAX'] = pred_SARIMAX

pred_df
```

#A We fit the model on the data from 1959 to the end of 2007.
#B The test set contains the values starting in 2008 to the 3rd quarter of 2009. There is a total of 7 values to predict.
#C This specifies that we predict the next timestep only.

With the predictions done, we can visualize which model has the lowest mean absolute percentage error or MAPE. The result is shown in figure 9.6.

```
def mape(y_true, y_pred):
    return np.mean(np.abs((y_true - y_pred) / y_true)) * 100

mape_last = mape(pred_df.actual, pred_df.pred_last_value)
mape_SARIMAX = mape(pred_df.actual, pred_df.pred_SARIMAX)

fig, ax = plt.subplots()

x = ['naive last value', 'SARIMAX']
y = [mape_last, mape_SARIMAX]

ax.bar(x, y, width=0.4)
ax.set_xlabel('Models')
ax.set_ylabel('MAPE (%)')
ax.set_ylim(0, 1)

for index, value in enumerate(y):
    plt.text(x=index, y=value + 0.05, s=str(round(value,2)), ha='center')

plt.tight_layout()
```

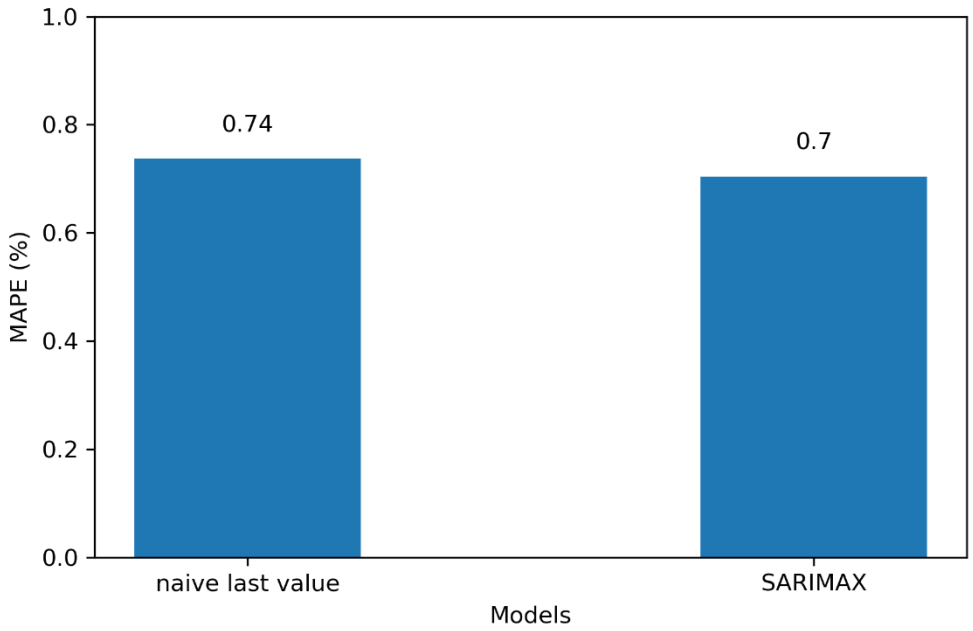


Figure 9.7 Mean absolute percentage error of the forecasts of each method. We see that the SARIMAX model only has a slightly smaller MAPE compared to the baseline. This highlights the importance of a baseline, as a MAPE of 0.70% is extremely good, but a naïve forecast achieves a MAPE of 0.74%, meaning that the SARIMAX model only has a small advantage.

Looking at figure 9.6, we see that the SARIMAX model is the winning model by only 0.04%. We can appreciate the importance of a baseline here, as both methods achieve an extremely low MAPE, showing that the SARIMAX model is only slightly better than simply predicting the last value. This is where the business context comes into play. In our case, since we are predicting the real GDP of the United States, a difference of 0.04% represents thousands of dollars. This difference might be relevant in this particular context, justifying the use of the SARIMAX model, even though it is only slightly better than the baseline.

9.3 Next steps

In this chapter, we covered the SARIMAX model, which allows us to include external variables to forecast our target time series.

The addition of exogenous variables comes with a caveat where if we need to predict many timesteps into the future, we must then also predict the exogenous variables, which can magnify the prediction error on the target. To avoid that, we must then only predict the next timestep.

Now, in considering exogenous variable to predict real GDP, we can also hypothesize that real GDP can be a predictor for other variables. For example, the variable `cpi` was a predictor for `realgdp`, but we could also show that `realgdp` can predict `cpi`.

In a situation where we wish to show that two variables varying in time can impact one another, we must use the **vector autoregression** model or **VAR**. This model allows for multivariate time series forecasting, unlike the SARIMAX model which is for univariate time series forecasting. In the next chapter, we will explore the VAR model in detail, and see that it can also be extended to become a **VARMA** model and a **VARMAX** model.

9.4 Exercises

9.4.1 Easy: Use all exogenous variables in a SARIMAX model to predict the real GDP

In this chapter, we limited the number of exogenous variables when forecasting for the real GDP.

This exercise is the occasion to fit a SARIMAX model using all exogenous variables and verify if we can achieve a better performance.

- Use all exogenous variables in the SARIMAX model.
- Perform residual analysis.
- Produce forecast for the last 7 timesteps in the dataset.
- Measure the MAPE. Is it better, worse, or identical to what was achieved with a limited number of exogenous variables?

9.5 Summary

- The SARIMAX model allows us to include external variables, also termed exogenous variables, to forecast our target.
- Transformations are applied only on the target variable, not the exogenous variables.
- If we wish to forecast multiple timesteps into the future, then the exogenous variables must also be forecast. This can magnify the errors on the final forecast. To avoid that, we must predict only the next timestep.