Chapter 14

Use and Remove Seasonality

Time series datasets can contain a seasonal component. This is a cycle that repeats over time, such as monthly or yearly. This repeating cycle may obscure the signal that we wish to model when forecasting, and in turn may provide a strong signal to our predictive models. In this tutorial, you will discover how to identify and correct for seasonality in time series data with Python.

After completing this tutorial, you will know:

- The definition of seasonality in time series and the opportunity it provides for forecasting with machine learning methods.
- How to use the difference method to create a seasonally adjusted time series of daily temperature data.
- How to model the seasonal component directly and explicitly subtract it from observations.

Let's get started.

14.1 Seasonality in Time Series

Time series data may contain seasonal variation. Seasonal variation, or seasonality, are cycles that repeat regularly over time.

> A repeating pattern within each year is known as seasonal variation, although the term is applied more generally to repeating patterns within any fixed period.
>
> — Page 6, *Introductory Time Series with R.*

A cycle structure in a time series may or may not be seasonal. If it consistently repeats at the same frequency, it is seasonal, otherwise it is not seasonal and is called a cycle.

14.1.1 Benefits to Machine Learning

Understanding the seasonal component in time series can improve the performance of modeling with machine learning. This can happen in two main ways:

- Clearer Signal: Identifying and removing the seasonal component from the time series can result in a clearer relationship between input and output variables.
- More Information: Additional information about the seasonal component of the time series can provide new information to improve model performance.

Both approaches may be useful on a project. Modeling seasonality and removing it from the time series may occur during data cleaning and preparation. Extracting seasonal information and providing it as input features, either directly or in summary form, may occur during feature extraction and feature engineering activities.

14.1.2 Types of Seasonality

There are many types of seasonality; for example:

- Time of Day.
- Daily.
- Weekly.
- Monthly.
- Yearly.

As such, identifying whether there is a seasonality component in your time series problem is subjective. The simplest approach to determining if there is an aspect of seasonality is to plot and review your data, perhaps at different scales and with the addition of trend lines.

14.1.3 Removing Seasonality

Once seasonality is identified, it can be modeled. The model of seasonality can be removed from the time series. This process is called Seasonal Adjustment, or Deseasonalizing. A time series where the seasonal component has been removed is called seasonal stationary. A time series with a clear seasonal component is referred to as non-stationary.

There are sophisticated methods to study and extract seasonality from time series in the field of Time Series Analysis. As we are primarily interested in predictive modeling and time series forecasting, we are limited to methods that can be developed on historical data and available when making predictions on new data. In this tutorial, we will look at two methods for making seasonal adjustments on a classical meteorological-type problem of daily temperatures with a strong additive seasonal component. Next, let's take a look at the dataset we will use in this tutorial.

14.2 Minimum Daily Temperatures Dataset

In this lesson, we will use the Minimum Daily Temperatures dataset as an example. This dataset describes the minimum daily temperatures over 10 years (1981-1990) in the city Melbourne, Australia. You can learn more about the dataset in Appendix A.2. Place the dataset in your current working directory with the filename `daily-minimum-temperatures.csv`.

14.3 Seasonal Adjustment with Differencing

A simple way to correct for a seasonal component is to use differencing. If there is a seasonal component at the level of one week, then we can remove it on an observation today by subtracting the value from last week. In the case of the Minimum Daily Temperatures dataset, it looks like we have a seasonal component each year showing swing from summer to winter.

We can subtract the daily minimum temperature from the same day last year to correct for seasonality. This would require special handling of February 29th in leap years and would mean that the first year of data would not be available for modeling. Below is an example of using the difference method on the daily data in Python.

```
# deseasonalize a time series using differencing
from pandas import Series
from matplotlib import pyplot
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
X = series.values
diff = list()
days_in_year = 365
for i in range(days_in_year, len(X)):
  value = X[i] - X[i - days_in_year]
  diff.append(value)
pyplot.plot(diff)
pyplot.show()
```

Listing 14.1: Deseasonalize Minimum Daily Temperatures dataset using differencing.

Running this example creates a new seasonally adjusted dataset and plots the result.

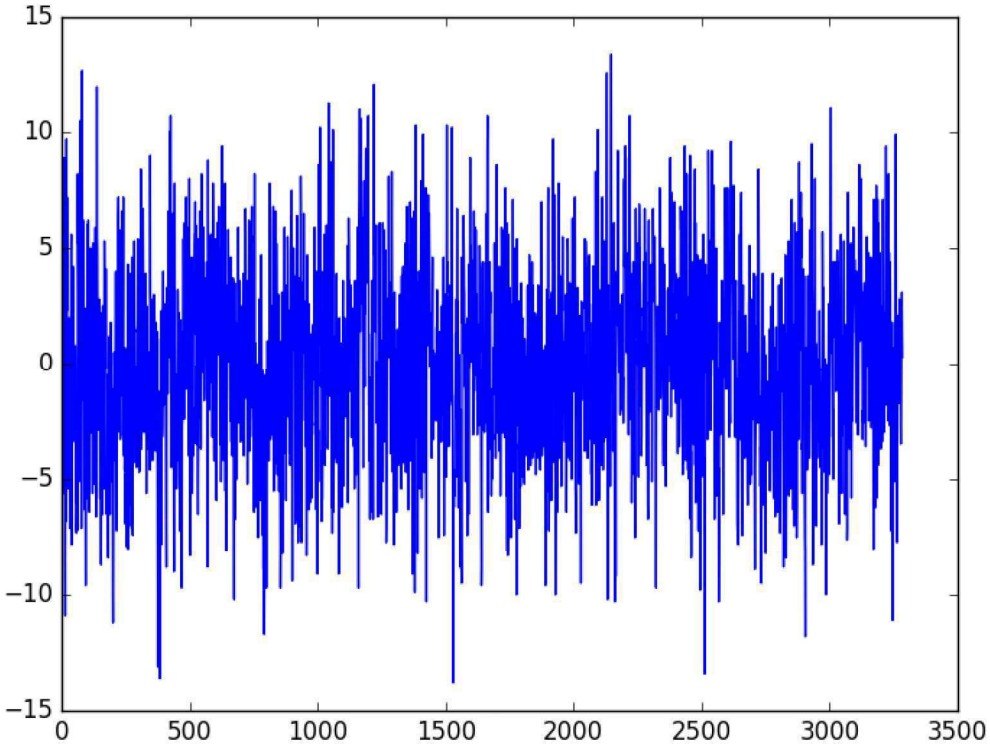

Figure 14.1: Line plot of the deseasonalized Minimum Daily Temperatures dataset using differencing.

There are two leap years in our dataset (1984 and 1988). They are not explicitly handled; this means that observations in March 1984 onwards the offset are wrong by one day, and after March 1988, the offsets are wrong by two days. One option is to update the code example to be leap-day aware.

Another option is to consider that the temperature within any given period of the year is probably stable. Perhaps over a few weeks. We can shortcut this idea and consider all temperatures within a calendar month to be stable. An improved model may be to subtract the average temperature from the same calendar month in the previous year, rather than the same day. We can start off by resampling the dataset to a monthly average minimum temperature.

```
# calculate and plot monthly average
from pandas import Series
from matplotlib import pyplot
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
resample = series.resample('M')
monthly_mean = resample.mean()
print(monthly_mean.head(13))
monthly_mean.plot()
pyplot.show()
```

Listing 14.2: Resampled Minimum Daily Temperatures dataset to monthly.

Running this example prints the first 13 months of average monthly minimum temperatures.

```
Date
1981-01-31   17.712903
1981-02-28   17.678571
1981-03-31   13.500000
1981-04-30   12.356667
1981-05-31    9.490323
1981-06-30    7.306667
1981-07-31    7.577419
1981-08-31    7.238710
1981-09-30   10.143333
1981-10-31   10.087097
1981-11-30   11.890000
1981-12-31   13.680645
1982-01-31   16.567742
```

Listing 14.3: Example output of first 13 months of rescaled Minimum Daily Temperatures dataset.

It also plots the monthly data, clearly showing the seasonality of the dataset.

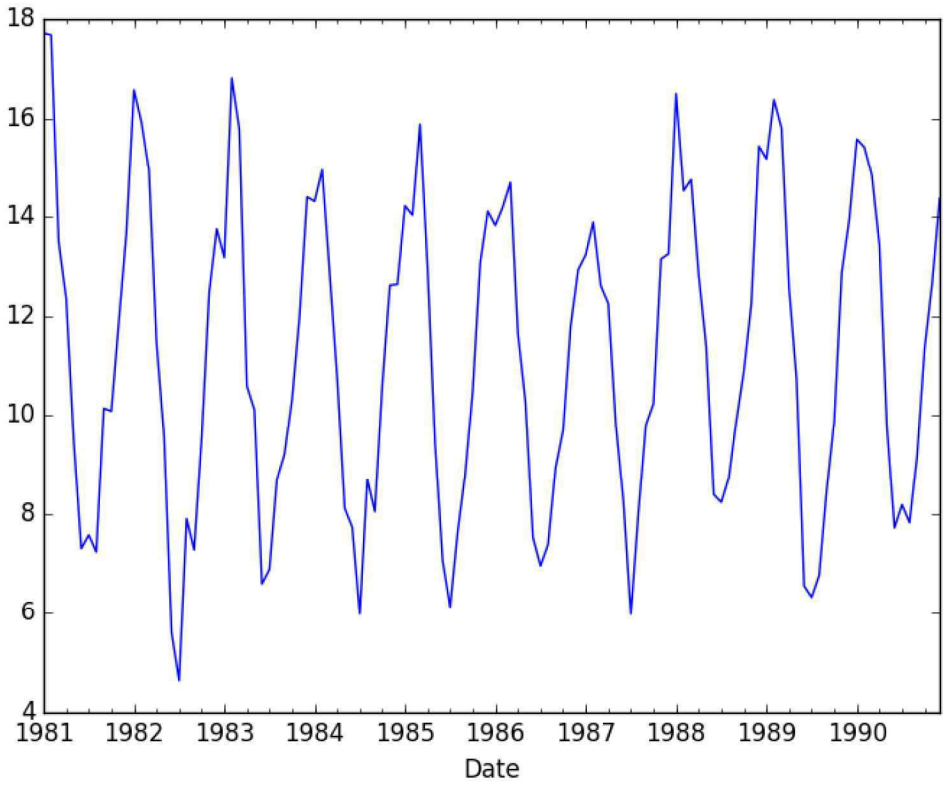

Figure 14.2: Line plot of the monthly Minimum Daily Temperatures dataset.

We can test the same differencing method on the monthly data and confirm that the seasonally adjusted dataset does indeed remove the yearly cycles.

```
# deseasonalize monthly data by differencing
from pandas import Series
from matplotlib import pyplot
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
resample = series.resample('M')
monthly_mean = resample.mean()
X = series.values
diff = list()
months_in_year = 12
for i in range(months_in_year, len(monthly_mean)):
  value = monthly_mean[i] - monthly_mean[i - months_in_year]
  diff.append(value)
pyplot.plot(diff)
pyplot.show()
```

Listing 14.4: Monthly Deseasonalized Minimum Daily Temperatures dataset using differencing.

Running the example creates a new seasonally adjusted monthly minimum temperature dataset, skipping the first year of data in order to create the adjustment. The adjusted dataset is then plotted.

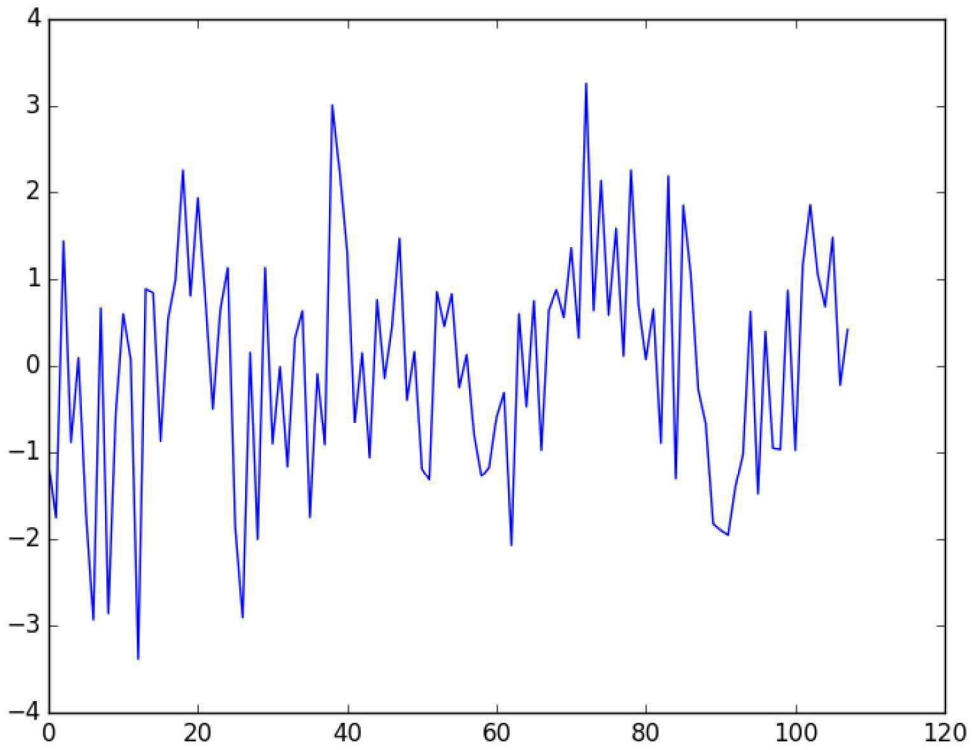

Figure 14.3: Line plot of the deseasonalized monthly Minimum Daily Temperatures dataset.

Next, we can use the monthly average minimum temperatures from the same month in the previous year to adjust the daily minimum temperature dataset. Again, we just skip the first

year of data, but the correction using the monthly rather than the daily data may be a more stable approach.

```
# deseasonalize a time series using month-based differencing
from pandas import Series
from matplotlib import pyplot
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
X = series.values
diff = list()
days_in_year = 365
for i in range(days_in_year, len(X)):
  month_str = str(series.index[i].year-1)+'-'+str(series.index[i].month)
  month_mean_last_year = series[month_str].mean()
  value = X[i] - month_mean_last_year
  diff.append(value)
pyplot.plot(diff)
pyplot.show()
```

Listing 14.5: Deseasonalized Minimum Daily Temperatures dataset using differencing at the month level.

Running the example again creates the seasonally adjusted dataset and plots the results. This example is robust to daily fluctuations in the previous year and to offset errors creeping in due to February 29 days in leap years.

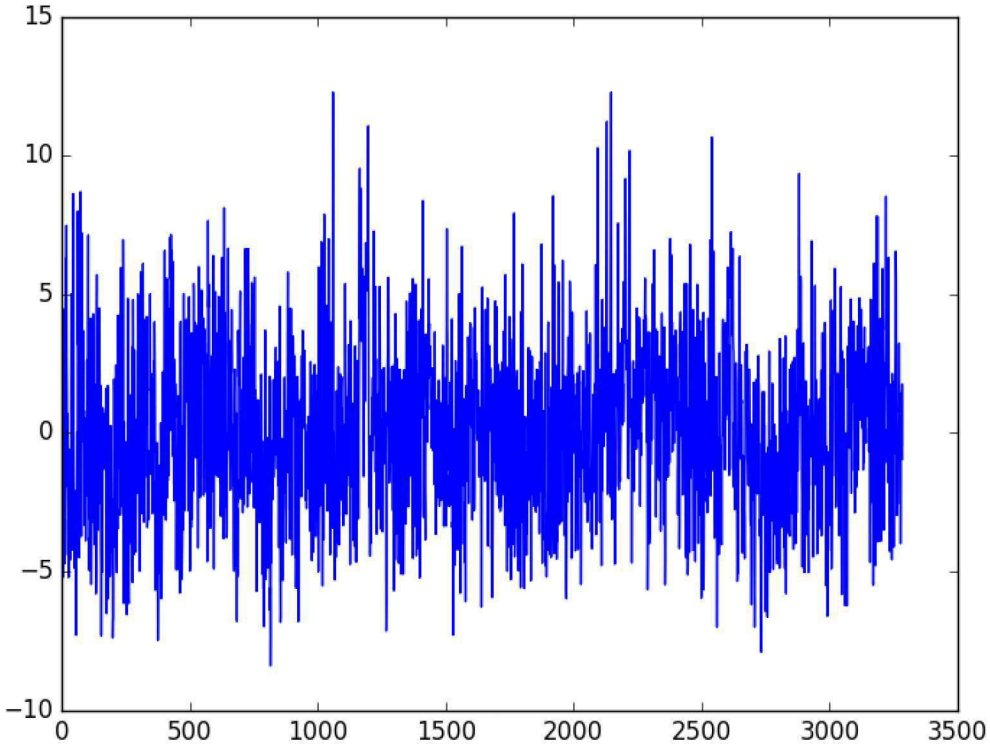

Figure 14.4: Line plot of the deseasonalized Minimum Daily Temperatures dataset using monthly data.

The edge of calendar months provides a hard boundary that may not make sense for temperature data. More flexible approaches that take the average from one week either side of the same date in the previous year may again be a better approach. Additionally, there is likely to be seasonality in temperature data at multiple scales that may be corrected for directly or indirectly, such as:

- Day level.
- Multiple day level, such as a week or weeks.
- Multiple week level, such as a month.
- Multiple month level, such as a quarter or season.

14.4 Seasonal Adjustment with Modeling

We can model the seasonal component directly, then subtract it from the observations. The seasonal component in a given time series is likely a sine wave over a generally fixed period and amplitude. This can be approximated easily using a curve-fitting method. A dataset can be

constructed with the time index of the sine wave as an input, or x-axis, and the observation as the output, or y-axis. For example:

```
Time Index, Observation
1, obs1
2, obs2
3, obs3
4, obs4
5, obs5
```

Listing 14.6: Example of time index as a feature.

Once fit, the model can then be used to calculate a seasonal component for any time index. In the case of the temperature data, the time index would be the day of the year. We can then estimate the seasonal component for the day of the year for any historical observations or any new observations in the future. The curve can then be used as a new input for modeling with supervised learning algorithms, or subtracted from observations to create a seasonally adjusted series.

Let's start off by fitting a curve to the Minimum Daily Temperatures dataset. The NumPy library provides the `polyfit()` function[1] that can be used to fit a polynomial of a chosen order to a dataset. First, we can create a dataset of time index (day in this case) to observation. We could take a single year of data or all the years. Ideally, we would try both and see which model resulted in a better fit. We could also smooth the observations using a moving average centered on each value. This too may result in a model with a better fit.

Once the dataset is prepared, we can create the fit by calling the `polyfit()` function passing the x-axis values (integer day of year), y-axis values (temperature observations), and the order of the polynomial. The order controls the number of terms, and in turn the complexity of the curve used to fit the data. Ideally, we want the simplest curve that describes the seasonality of the dataset. For consistent sine wave-like seasonality, a 4th order or 5th order polynomial will be sufficient. In this case, I chose an order of 4 by trial and error. The resulting model takes the form:

$$y = (x^4 \times b1) + (x^3 \times b2) + (x^2 \times b3) + (x^1 \times b4) + b5 \tag{14.1}$$

Where y is the fit value, x is the time index (day of the year), and `b1` to `b5` are the coefficients found by the curve-fitting optimization algorithm. Once fit, we will have a set of coefficients that represent our model. We can then use this model to calculate the curve for one observation, one year of observations, or the entire dataset. The complete example is listed below.

```
# model seasonality with a polynomial model
from pandas import Series
from matplotlib import pyplot
from numpy import polyfit
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
# fit polynomial: x^2*b1 + x*b2 + ... + bn
X = [i%365 for i in range(0, len(series))]
y = series.values
degree = 4
coef = polyfit(X, y, degree)
print('Coefficients: %s' % coef)
# create curve
```

[1] https://docs.scipy.org/doc/numpy/reference/generated/numpy.polyfit.html

```
curve = list()
for i in range(len(X)):
  value = coef[-1]
  for d in range(degree):
    value += X[i]**(degree-d) * coef[d]
  curve.append(value)
# plot curve over original data
pyplot.plot(series.values)
pyplot.plot(curve, color='red', linewidth=3)
pyplot.show()
```

Listing 14.7: Nonlinear model of seasonality in the Minimum Daily Temperatures dataset.

Running the example creates the dataset, fits the curve, predicts the value for each day in the dataset, and then plots the resulting seasonal model over the top of the original dataset. One limitation of this model is that it does not take into account of leap days, adding small offset noise that could easily be corrected with an update to the approach. For example, we could just remove the two February 29 observations from the dataset when creating the seasonal model.

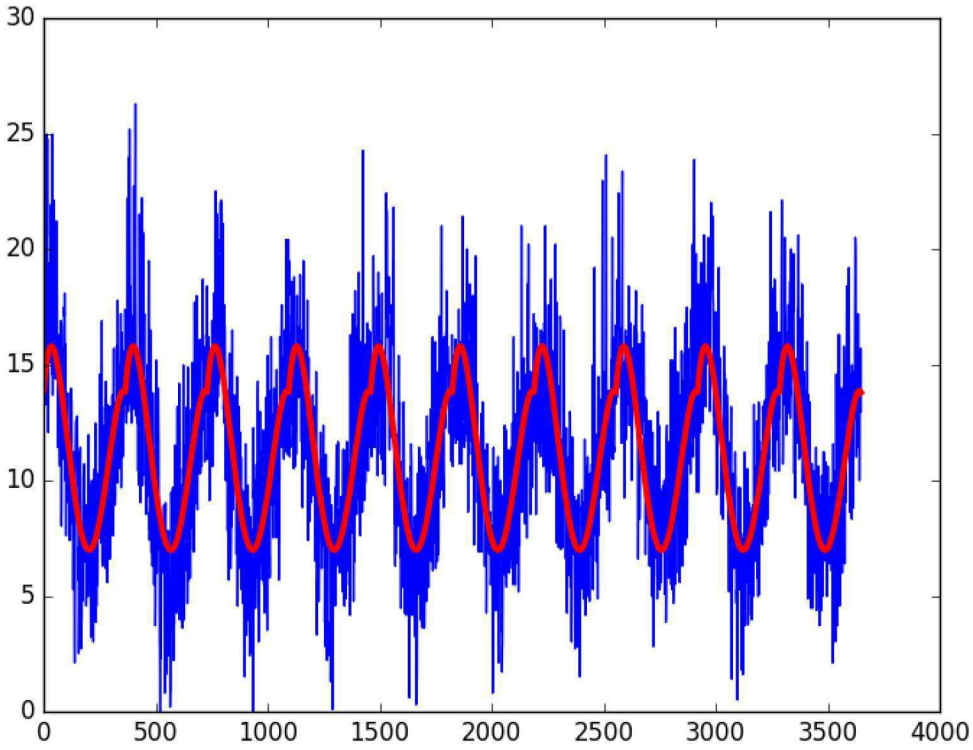

Figure 14.5: Line plot of the Minimum Daily Temperatures dataset (blue) and a nonlinear model of the seasonality (red).

The curve appears to be a good fit for the seasonal structure in the dataset. We can now

use this model to create a seasonally adjusted version of the dataset. The complete example is listed below.

```
# deseasonalize by differencing with a polynomial model
from pandas import Series
from matplotlib import pyplot
from numpy import polyfit
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
# fit polynomial: x^2*b1 + x*b2 + ... + bn
X = [i%365 for i in range(0, len(series))]
y = series.values
degree = 4
coef = polyfit(X, y, degree)
# create curve
curve = list()
for i in range(len(X)):
  value = coef[-1]
  for d in range(degree):
    value += X[i]**(degree-d) * coef[d]
  curve.append(value)
# create seasonally adjusted
values = series.values
diff = list()
for i in range(len(values)):
  value = values[i] - curve[i]
  diff.append(value)
pyplot.plot(diff)
pyplot.show()
```

Listing 14.8: Deseasonalized Minimum Daily Temperatures dataset using a nonlinear model.

Running the example subtracts the values predicted by the seasonal model from the original observations. The The seasonally adjusted dataset is then plotted.

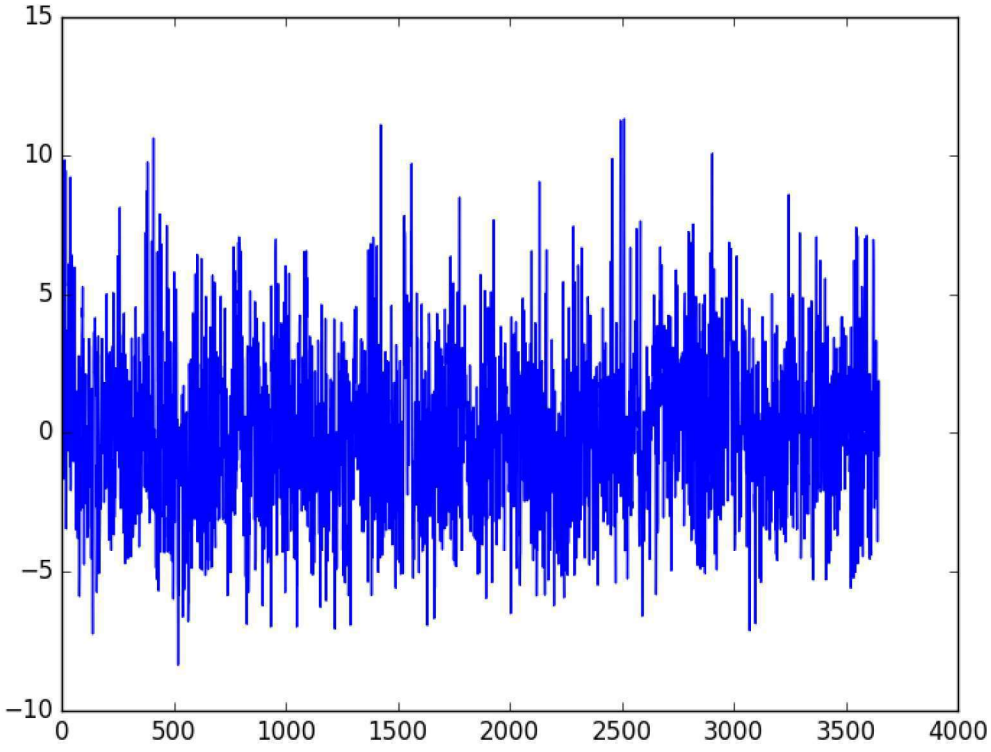

Figure 14.6: Line plot of the deseasonalized Minimum Daily Temperatures dataset using a nonlinear model.

14.5 Summary

In this tutorial, you discovered how to create seasonally adjusted time series datasets in Python. Specifically, you learned:

- The importance of seasonality in time series and the opportunities for data preparation and feature engineering it provides.
- How to use the difference method to create a seasonally adjusted time series.
- How to model the seasonal component directly and subtract it from observations.

14.5.1 Next

In the next lesson you will discover how to identify and test for stationarity in time series data.

Chapter 15

Stationarity in Time Series Data

Time series is different from more traditional classification and regression predictive modeling problems. The temporal structure adds an order to the observations. This imposed order means that important assumptions about the consistency of those observations needs to be handled specifically. For example, when modeling, there are assumptions that the summary statistics of observations are consistent. In time series terminology, we refer to this expectation as the time series being stationary. These assumptions can be easily violated in time series by the addition of a trend, seasonality, and other time-dependent structures. In this tutorial, you will discover how to check if your time series is stationary with Python. After completing this tutorial, you will know:

- How to identify obvious stationary and non-stationary time series using line plot.
- How to spot-check summary statistics like mean and variance for a change over time.
- How to use statistical tests with statistical significance to check if a time series is stationary.

Let's get started.

15.1 Stationary Time Series

The observations in a stationary time series are not dependent on time. Time series are stationary if they do not have trend or seasonal effects. Summary statistics calculated on the time series are consistent over time, like the mean or the variance of the observations. When a time series is stationary, it can be easier to model. Statistical modeling methods assume or require the time series to be stationary to be effective. Below is an example of the Daily Female Births dataset that is stationary. You can learn more about the dataset in Appendix A.4.

```
# load time series data
from pandas import Series
from matplotlib import pyplot
series = Series.from_csv('daily-total-female-births.csv', header=0)
series.plot()
pyplot.show()
```

Listing 15.1: Load and plot Daily Female Births dataset.

Running the example creates the following plot.

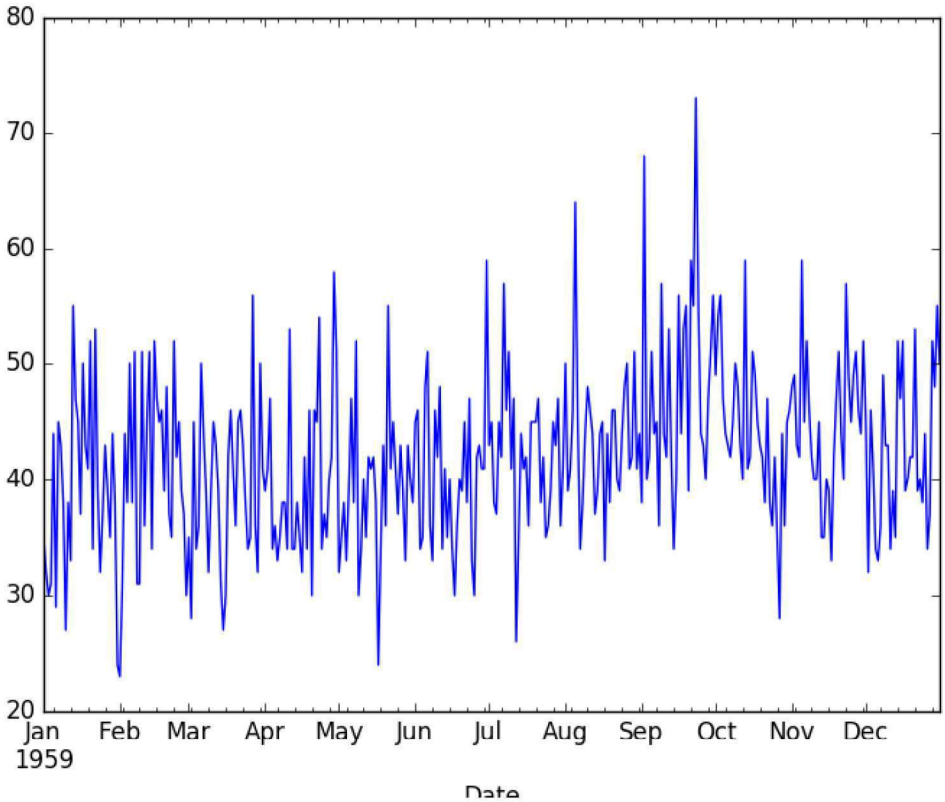

Figure 15.1: Line plot of the stationary Daily Female Births time series dataset.

15.2 Non-Stationary Time Series

Observations from a non-stationary time series show seasonal effects, trends, and other structures that depend on the time index. Summary statistics like the mean and variance do change over time, providing a drift in the concepts a model may try to capture. Classical time series analysis and forecasting methods are concerned with making non-stationary time series data stationary by identifying and removing trends and removing stationary effects. Below is an example of the Airline Passengers dataset that is non-stationary, showing both trend and seasonal components. You can learn more about the dataset in Appendix A.5.

```
# load time series data
from pandas import Series
from matplotlib import pyplot
series = Series.from_csv('airline-passengers.csv', header=0)
series.plot()
pyplot.show()
```

Listing 15.2: Load and plot the Airline Passengers dataset.

Running the example creates the following plot.

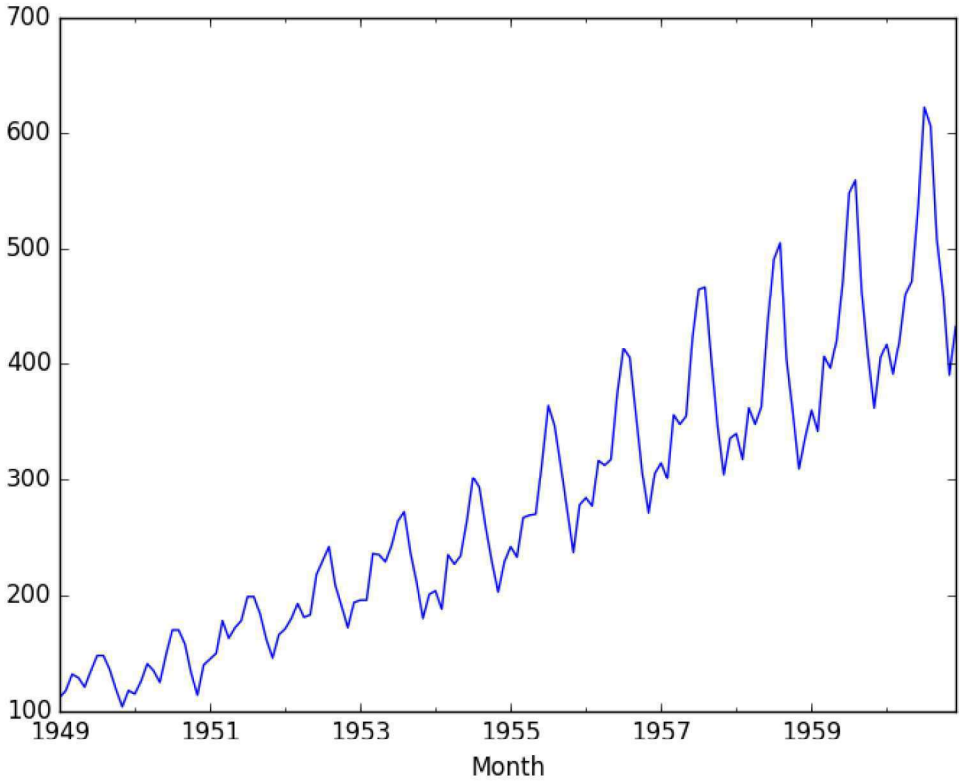

Figure 15.2: Line plot of the non-stationary Airline Passengers time series dataset.

15.3 Types of Stationary Time Series

The notion of stationarity comes from the theoretical study of time series and it is a useful abstraction when forecasting. There are some finer-grained notions of stationarity that you may come across if you dive deeper into this topic. They are:

- **Stationary Process**: A process that generates a stationary series of observations.
- **Stationary Model**: A model that describes a stationary series of observations.
- **Trend Stationary**: A time series that does not exhibit a trend.
- **Seasonal Stationary**: A time series that does not exhibit seasonality.
- **Strictly Stationary**: A mathematical definition of a stationary process, specifically that the joint distribution of observations is invariant to time shift.

15.4 Stationary Time Series and Forecasting

Should you make your time series stationary? Generally, yes. If you have clear trend and seasonality in your time series, then model these components, remove them from observations, then train models on the residuals.

> If we fit a stationary model to data, we assume our data are a realization of a stationary process. So our first step in an analysis should be to check whether there is any evidence of a trend or seasonal effects and, if there is, remove them.
>
> — Page 122, *Introductory Time Series with R.*

Statistical time series methods and even modern machine learning methods will benefit from the clearer signal in the data. But...

We turn to machine learning methods when the classical methods fail. When we want more or better results. We cannot know how to best model unknown nonlinear relationships in time series data and some methods may result in better performance when working with non-stationary observations or some mixture of stationary and non-stationary views of the problem.

The suggestion here is to treat properties of a time series being stationary or not as another source of information that can be used in feature engineering and feature selection on your time series problem when using machine learning methods.

15.5 Checks for Stationarity

There are many methods to check whether a time series (direct observations, residuals, otherwise) is stationary or non-stationary.

- **Look at Plots**: You can review a time series plot of your data and visually check if there are any obvious trends or seasonality.
- **Summary Statistics**: You can review the summary statistics for your data for seasons or random partitions and check for obvious or significant differences.
- **Statistical Tests**: You can use statistical tests to check if the expectations of stationarity are met or have been violated.

Above, we have already introduced the Daily Female Births and Airline Passengers datasets as stationary and non-stationary respectively with plots showing an obvious lack and presence of trend and seasonality components. Next, we will look at a quick and dirty way to calculate and review summary statistics on our time series dataset for checking to see if it is stationary.

15.6 Summary Statistics

A quick and dirty check to see if your time series is non-stationary is to review summary statistics. You can split your time series into two (or more) partitions and compare the mean and variance of each group. If they differ and the difference is statistically significant, the time series is likely non-stationary. Next, let's try this approach on the Daily Births dataset.

15.6.1 Daily Births Dataset

Because we are looking at the mean and variance, we are assuming that the data conforms to a Gaussian (also called the bell curve or normal) distribution. We can also quickly check this by eyeballing a histogram of our observations.

```
# plot a histogram of a time series
from pandas import Series
from matplotlib import pyplot
series = Series.from_csv('daily-total-female-births.csv', header=0)
series.hist()
pyplot.show()
```

Listing 15.3: Create a histogram plot of the Daily Female Births dataset.

Running the example plots a histogram of values from the time series. We clearly see the bell curve-like shape of the Gaussian distribution, perhaps with a longer right tail.

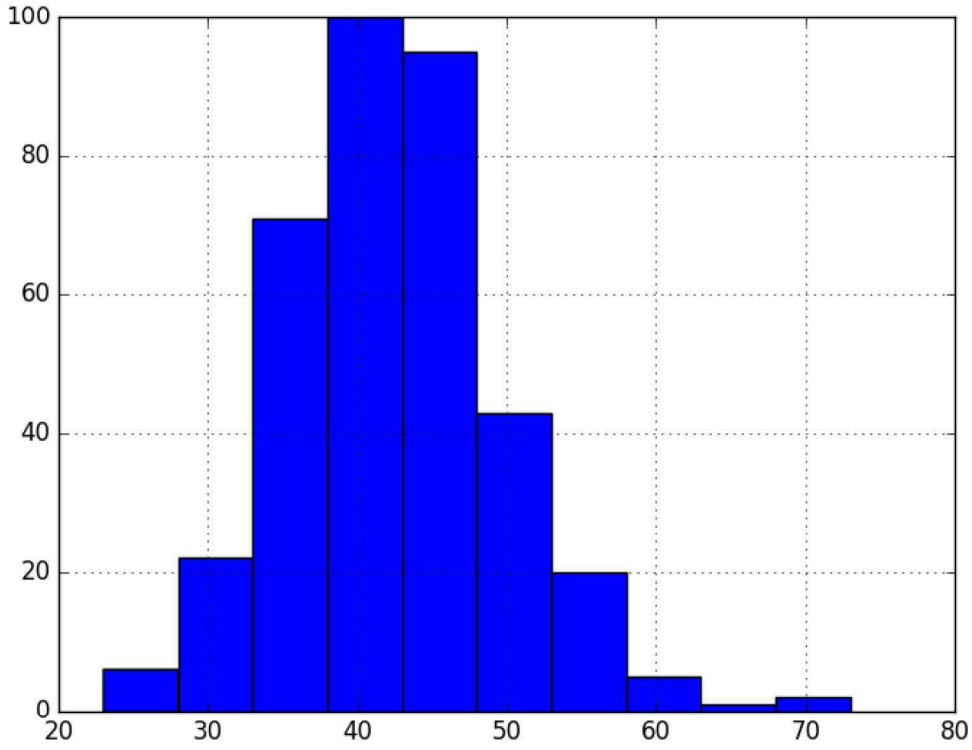

Figure 15.3: Histogram plot of the Daily Female Births dataset.

Next, we can split the time series into two contiguous sequences. We can then calculate the mean and variance of each group of numbers and compare the values.

```
# calculate statistics of partitioned time series data
from pandas import Series
series = Series.from_csv('daily-total-female-births.csv', header=0)
```

```
X = series.values
split = int(len(X) / 2)
X1, X2 = X[0:split], X[split:]
mean1, mean2 = X1.mean(), X2.mean()
var1, var2 = X1.var(), X2.var()
print('mean1=%f, mean2=%f' % (mean1, mean2))
print('variance1=%f, variance2=%f' % (var1, var2))
```

Listing 15.4: Summary statistics of the Daily Female Births dataset.

Running this example shows that the mean and variance values are different, but in the same ball-park.

```
mean1=39.763736, mean2=44.185792
variance1=49.213410, variance2=48.708651
```

Listing 15.5: Example output of summary statistics of the Daily Female Births dataset.

Next, let's try the same trick on the Airline Passengers dataset.

15.6.2 Airline Passengers Dataset

Cutting straight to the chase, we can split our dataset and calculate the mean and variance for each group.

```
# calculate statistics of partitioned time series data
from pandas import Series
series = Series.from_csv('airline-passengers.csv', header=0)
X = series.values
split = int(len(X) / 2)
X1, X2 = X[0:split], X[split:]
mean1, mean2 = X1.mean(), X2.mean()
var1, var2 = X1.var(), X2.var()
print('mean1=%f, mean2=%f' % (mean1, mean2))
print('variance1=%f, variance2=%f' % (var1, var2))
```

Listing 15.6: Summary statistics of the Airline Passengers dataset.

Running the example, we can see the mean and variance look very different. We have a non-stationary time series.

```
mean1=182.902778, mean2=377.694444
variance1=2244.087770, variance2=7367.962191
```

Listing 15.7: Example output of summary statistics of the Airline Passengers dataset.

Well, maybe. Let's take one step back and check if assuming a Gaussian distribution makes sense in this case by plotting the values of the time series as a histogram.

```
# plot a histogram of a time series
from pandas import Series
from matplotlib import pyplot
series = Series.from_csv('airline-passengers.csv', header=0)
series.hist()
pyplot.show()
```

Listing 15.8: Create a histogram plot of the Airline Passengers dataset.

Running the example shows that indeed the distribution of values does not look like a Gaussian, therefore the mean and variance values are less meaningful. This squashed distribution of the observations may be another indicator of a non-stationary time series.

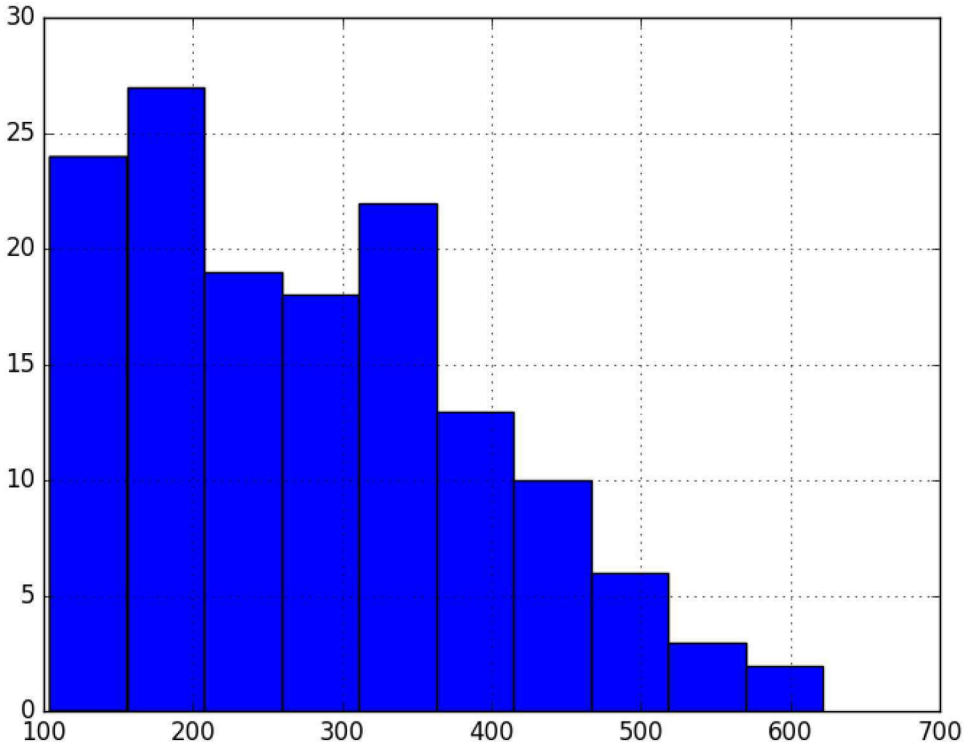

Figure 15.4: Histogram plot of the Airline Passengers dataset.

Reviewing the plot of the time series again, we can see that there is an obvious seasonality component, and it looks like the seasonality component is growing. This may suggest an exponential growth from season to season. A log transform can be used to flatten out exponential change back to a linear relationship. Below is the same histogram with a log transform of the time series.

```
# histogram and line plot of log transformed time series
from pandas import Series
from matplotlib import pyplot
from numpy import log
series = Series.from_csv('airline-passengers.csv', header=0)
X = series.values
X = log(X)
pyplot.hist(X)
pyplot.show()
pyplot.plot(X)
pyplot.show()
```

Listing 15.9: Create a histogram plot of the log-transformed Airline Passengers dataset.

Running the example, we can see the more familiar Gaussian-like or Uniform-like distribution of values.

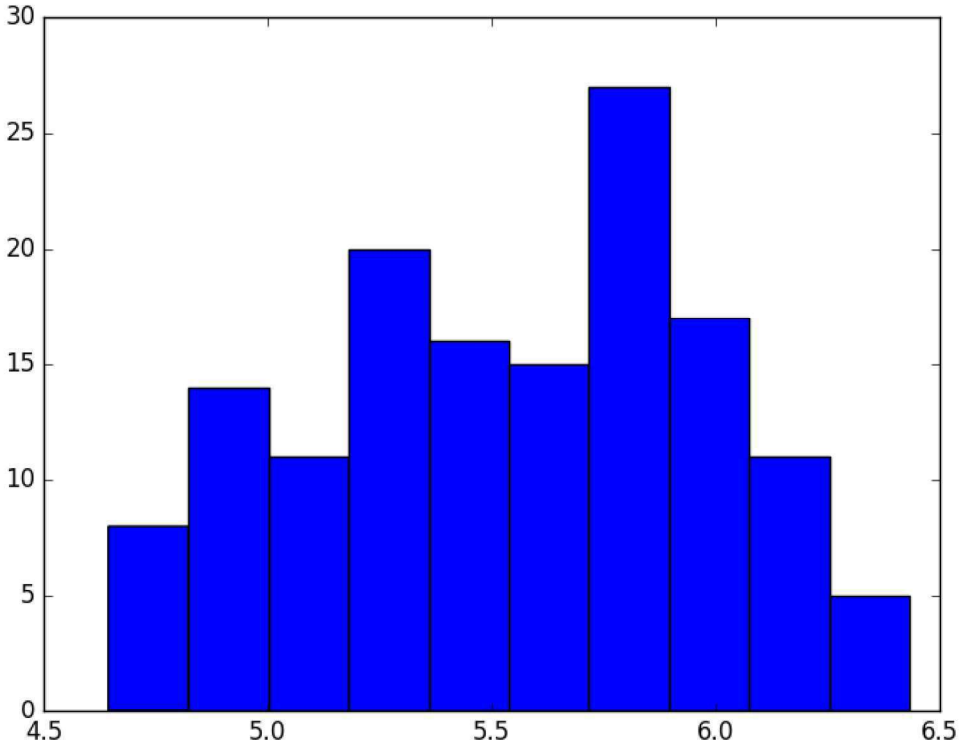

Figure 15.5: Histogram plot of the log-transformed Airline Passengers dataset.

We also create a line plot of the log transformed data and can see the exponential growth seems diminished (compared to the line plot of the dataset in the Appendix), but we still have a trend and seasonal elements.

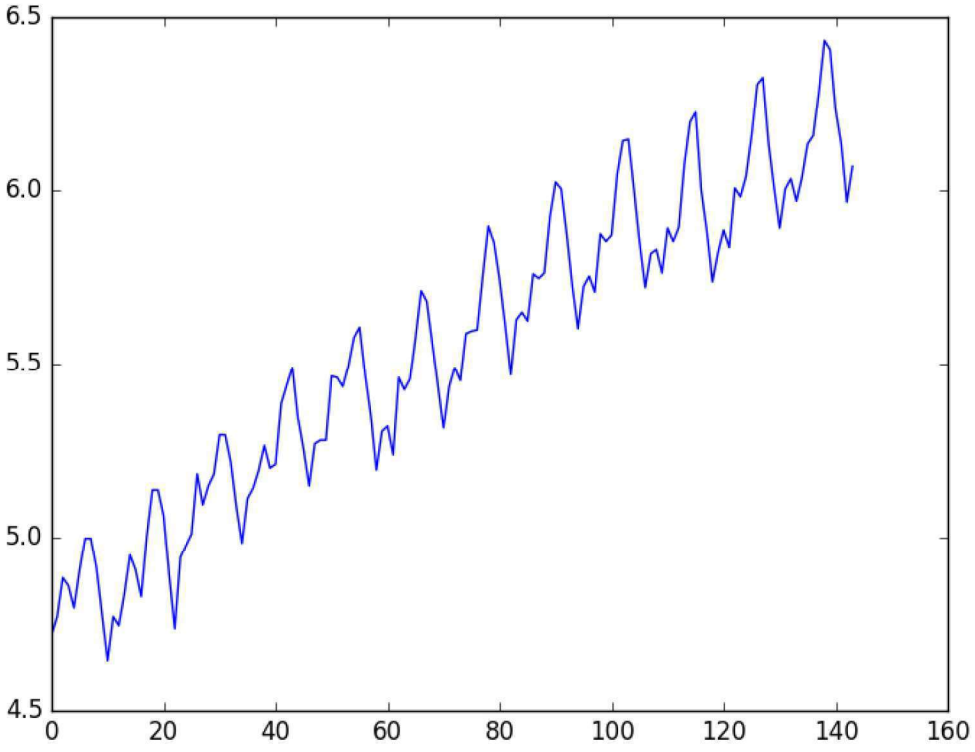

Figure 15.6: Line plot of the log-transformed Airline Passengers dataset.

We can now calculate the mean and standard deviation of the values of the log transformed dataset.

```
# calculate statistics of partitioned log transformed time series data
from pandas import Series
from matplotlib import pyplot
from numpy import log
series = Series.from_csv('airline-passengers.csv', header=0)
X = series.values
X = log(X)
split = int(len(X) / 2)
X1, X2 = X[0:split], X[split:]
mean1, mean2 = X1.mean(), X2.mean()
var1, var2 = X1.var(), X2.var()
print('mean1=%f, mean2=%f' % (mean1, mean2))
print('variance1=%f, variance2=%f' % (var1, var2))
```

Listing 15.10: Summary statistics of the log-transformed Airline Passengers dataset.

Running the examples shows mean and standard deviation values for each group that are again similar, but not identical. Perhaps, from these numbers alone, we would say the time series is stationary, but we strongly believe this to not be the case from reviewing the line plot.

```
mean1=5.175146, mean2=5.909206
```

```
variance1=0.068375, variance2=0.049264
```

Listing 15.11: Example output of summary statistics of the log-transformed Airline Passengers dataset.

This is a quick and dirty method that may be easily fooled. We can use a statistical test to check if the difference between two samples of Gaussian random variables is real or a statistical fluke. We could explore statistical significance tests, like the Student t-test, but things get tricky because of the serial correlation between values. In the next section, we will use a statistical test designed to explicitly comment on whether a univariate time series is stationary.

15.7 Augmented Dickey-Fuller test

Statistical tests make strong assumptions about your data. They can only be used to inform the degree to which a null hypothesis can be accepted or rejected. The result must be interpreted for a given problem to be meaningful. Nevertheless, they can provide a quick check and confirmatory evidence that your time series is stationary or non-stationary.

The Augmented Dickey-Fuller test is a type of statistical test called a unit root test[1]. The intuition behind a unit root test is that it determines how strongly a time series is defined by a trend.

There are a number of unit root tests and the Augmented Dickey-Fuller may be one of the more widely used. It uses an autoregressive model and optimizes an information criterion across multiple different lag values. The null hypothesis of the test is that the time series can be represented by a unit root, that it is not stationary (has some time-dependent structure). The alternate hypothesis (rejecting the null hypothesis) is that the time series is stationary.

- **Null Hypothesis (H0)**: If accepted, it suggests the time series has a unit root, meaning it is non-stationary. It has some time dependent structure.
- **Alternate Hypothesis (H1)**: The null hypothesis is rejected; it suggests the time series does not have a unit root, meaning it is stationary. It does not have time-dependent structure.

We interpret this result using the p-value from the test. A p-value below a threshold (such as 5% or 1%) suggests we reject the null hypothesis (stationary), otherwise a p-value above the threshold suggests we accept the null hypothesis (non-stationary).

- **p-value $>$ 0.05**: Accept the null hypothesis (H0), the data has a unit root and is non-stationary.
- **p-value $\leq$ 0.05**: Reject the null hypothesis (H0), the data does not have a unit root and is stationary.

Below is an example of calculating the Augmented Dickey-Fuller test on the Daily Female Births dataset. The Statsmodels library provides the `adfuller()` function[2] that implements the test.

[1] https://en.wikipedia.org/wiki/Augmented_Dickey%E2%80%93Fuller_test

[2] http://statsmodels.sourceforge.net/devel/generated/statsmodels.tsa.stattools.adfuller.html

```
# calculate stationarity test of time series data
from pandas import Series
from statsmodels.tsa.stattools import adfuller
series = Series.from_csv('daily-total-female-births.csv', header=0)
X = series.values
result = adfuller(X)
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])
print('Critical Values:')
for key, value in result[4].items():
  print('\t%s: %.3f' % (key, value))
```

Listing 15.12: Augmented Dickey-Fuller test on the Daily Female Births dataset.

Running the example prints the test statistic value of -4. The more negative this statistic, the more likely we are to reject the null hypothesis (we have a stationary dataset). As part of the output, we get a look-up table to help determine the ADF statistic. We can see that our statistic value of -4 is less than the value of -3.449 at 1%.

This suggests that we can reject the null hypothesis with a significance level of less than 1% (i.e. a low probability that the result is a statistical fluke). Rejecting the null hypothesis means that the process has no unit root, and in turn that the time series is stationary or does not have time-dependent structure.

```
ADF Statistic: -4.808291
p-value: 0.000052
Critical Values:
  5%: -2.870
  1%: -3.449
  10%: -2.571
```

Listing 15.13: Example output of the Augmented Dickey-Fuller test on the Daily Female Births dataset.

We can perform the same test on the Airline Passenger dataset.

```
# calculate stationarity test of time series data
from pandas import Series
from statsmodels.tsa.stattools import adfuller
series = Series.from_csv('airline-passengers.csv', header=0)
X = series.values
result = adfuller(X)
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])
print('Critical Values:')
for key, value in result[4].items():
  print('\t%s: %.3f' % (key, value))
```

Listing 15.14: Augmented Dickey-Fuller test on the Airline Passengers dataset.

Running the example gives a different picture than the above. The test statistic is positive, meaning we are much less likely to reject the null hypothesis (it looks non-stationary). Comparing the test statistic to the critical values, it looks like we would have to accept the null hypothesis that the time series is non-stationary and does have time-dependent structure.

```
ADF Statistic: 0.815369
```

```
p-value: 0.991880
Critical Values:
  5%: -2.884
  1%: -3.482
  10%: -2.579
```

Listing 15.15: Example output of the Augmented Dickey-Fuller test on the Airline Passengers dataset.

Let's log transform the dataset again to make the distribution of values more linear and better meet the expectations of this statistical test.

```
# calculate stationarity test of log transformed time series data
from pandas import Series
from statsmodels.tsa.stattools import adfuller
from numpy import log
series = Series.from_csv('airline-passengers.csv', header=0)
X = series.values
X = log(X)
result = adfuller(X)
print('ADF Statistic: %f' % result[0])
print('p-value: %f' % result[1])
for key, value in result[4].items():
  print('\t%s: %.3f' % (key, value))
```

Listing 15.16: Augmented Dickey-Fuller test on the log-transformed Airline Passengers dataset.

Running the example shows a negative value for the test statistic. We can see that the value is larger than the critical values, again, meaning that we can accept the null hypothesis and in turn that the time series is non-stationary.

```
ADF Statistic: -1.717017
p-value: 0.422367
  5%: -2.884
  1%: -3.482
  10%: -2.579
```

Listing 15.17: Example output of the Augmented Dickey-Fuller test on the log-transformed Airline Passengers dataset.

15.8 Summary

In this tutorial, you discovered how to check if your time series is stationary with Python. Specifically, you learned:

- The importance of time series data being stationary for use with statistical modeling methods and even some modern machine learning methods.
- How to use line plots and basic summary statistics to check if a time series is stationary.
- How to calculate and interpret statistical significance tests to check if a time series is stationary.