Chapter 1

Python Environment

The Python ecosystem is growing and may become the dominant platform for applied machine learning. The primary rationale for adopting Python for time series forecasting is because it is a general-purpose programming language that you can use both for R&D and in production. In this lesson, you will discover the Python ecosystem for time series forecasting. After reading this lesson, you will know:

- The three standard Python libraries that are critical for time series forecasting.
- How to install and setup the Python and SciPy environment for development.
- How to confirm your environment is working correctly and ready for time series forecasting.

Let's get started.

1.1 Why Python?

Python is a general-purpose interpreted programming language (unlike R or Matlab). It is easy to learn and use primarily because the language focuses on readability. It is a popular language in general, consistently appearing in the top 10 programming languages in surveys on StackOverflow (for example, the 2015 survey results[1]).

Python is a dynamic language and is well suited to interactive development and quick prototyping with the power to support the development of large applications. This is a simple and very important consideration. It means that you can perform your research and development (figuring out what models to use) in the same programming language that you use in operations, greatly simplifying the transition from development to operations.

Python is also widely used for machine learning and data science because of the excellent library support. It has quickly become one of the dominant platforms for machine learning and data science practitioners and is in greater demand than even the R platform by employers (see the graph below).

[1] http://stackoverflow.com/research/developer-survey-2016

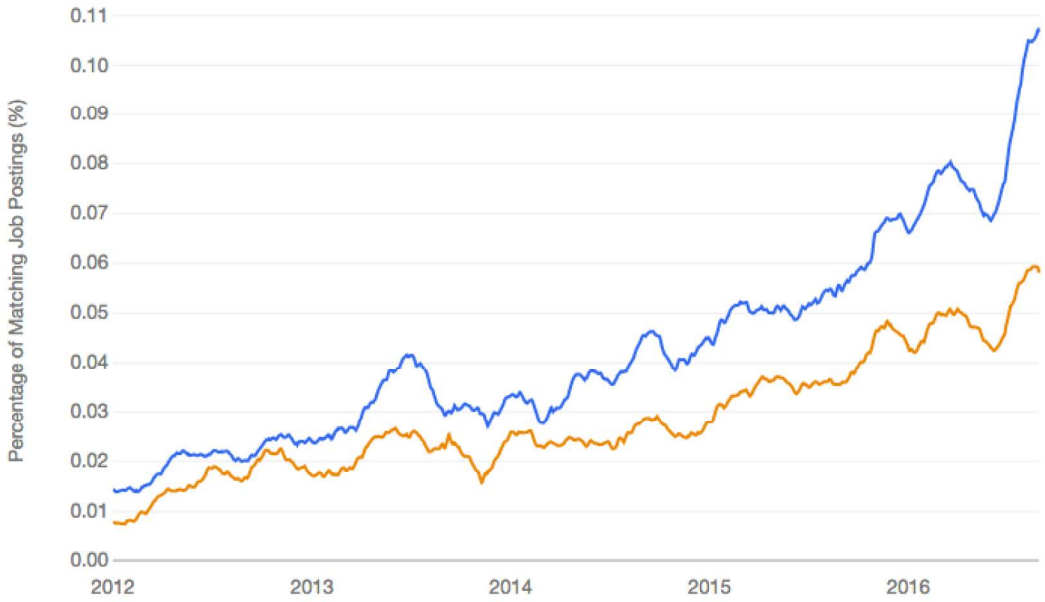

Figure 1.1: Plot of Python machine learning jobs (blue) compared to R machine learning jobs from indeed.com.

1.2 Python Libraries for Time Series

SciPy is an ecosystem of Python libraries for mathematics, science, and engineering[2]. It is an add-on to Python that you will need for time series forecasting. Two SciPy libraries provide a foundation for most others; they are NumPy[3] for providing efficient array operations and Matplotlib[4] for plotting data. There are three higher-level SciPy libraries that provide the key features for time series forecasting in Python. They are Pandas, Statsmodels, and scikit-learn for data handling, time series modeling, and machine learning respectively. Let's take a closer look at each in turn.

1.2.1 Library: Pandas

The Pandas library provides high-performance tools for loading and handling data in Python. It is built upon and requires the SciPy ecosystem and uses primarily NumPy arrays under the covers but provides convenient and easy to use data structures like `DataFrame` and `Series` for representing data. Pandas provides a special focus on support for time series data. Key features relevant for time series forecasting in Pandas include:

- The `Series` object for representing a univariate time series.

[2] https://www.scipy.org
[3] http://www.numpy.org
[4] http://matplotlib.org

- Explicit handling of date-time indexes in data and date-time ranges.
- Transforms such as shifting, lagging, and filling.
- Resampling methods such as up-sampling, down-sampling, and aggregation.

For further information on Pandas, see:

- Pandas homepage:
  http://pandas.pydata.org/
- Pandas Support for Time Series:
  http://pandas.pydata.org/pandas-docs/stable/timeseries.html

1.2.2 Library: Statsmodels

The Statsmodels library provides tools for statistical modeling. It is built upon and requires the SciPy ecosystem and supports data in the form of NumPy arrays and Pandas `Series` objects. It provides a suite of statistical test and modeling methods, as well as tools dedicated to time series analysis that can also be used for forecasting. Key features of Statsmodels relevant to time series forecasting include:

- Statistical tests for stationarity such as the Augmented Dickey-Fuller unit root test.
- Time series analysis plots such as autocorrelation function (ACF) and partial autocorrelation function (PACF).
- Linear time series models such as autoregression (AR), moving average (MA), autoregressive moving average (ARMA), and autoregressive integrated moving average (ARIMA).

For further information on Statsmodels, see:

- Statsmodels homepage:
  http://statsmodels.sourceforge.net/
- Statsmodels support for time series:
  http://statsmodels.sourceforge.net/stable/tsa.html

1.2.3 Library: scikit-learn

The scikit-learn library is how you can develop and practice machine learning in Python. The focus of the library is machine learning algorithms for classification, regression, clustering, and more. It also provides tools for related tasks such as evaluating models, tuning parameters, and pre-processing data. Key features relevant for time series forecasting in scikit-learn include:

- The suite of data preparation tools, such as scaling and imputing data.
- The suite of machine learning algorithms that could be used to model data and make predictions.

- The resampling methods for estimating the performance of a model on unseen data, specifically the `TimeSeriesSplit` class (covered in Chapter 16).

For further information on scikit-learn, see:

- Scikit-learn homepage:
  http://scikit-learn.org/

1.3 Python Ecosystem Installation

This section will provide you general advice for setting up your Python environment for time series forecasting. We will cover:

1. Automatic installation with Anaconda.
2. Manual installation with your platform's package management.
3. Confirmation of the installed environment.

If you already have a functioning Python environment, skip to the confirmation step to check if your software libraries are up-to-date. Let's dive in.

1.3.1 Automatic Installation

If you are not confident at installing software on your machine or you're on Microsoft Windows, there is an easy option for you. There is a distribution called Anaconda Python that you can download and install for free. It supports the three main platforms of Microsoft Windows, Mac OS X, and Linux. It includes Python, SciPy, and scikit-learn: everything you need to learn, practice, and use time series forecasting with the Python Environment.

You can get started with Anaconda Python here:

- Anaconda Install:
  https://docs.continuum.io/anaconda/install

1.3.2 Manual Installation

There are multiple ways to install the Python ecosystem specific to your platform. In this section, we cover how to install the Python ecosystem for time series forecasting.

How To Install Python

The first step is to install Python. I prefer to use and recommend Python 2.7 or Python 3.5. Installation of Python will be specific to your platform. For instructions see:

- Downloading Python in the Python Beginners Guide:
  https://wiki.python.org/moin/BeginnersGuide/Download

On Mac OS X with `macports`, you can type:

```
sudo port install python35
sudo port select --set python python35
sudo port select --set python3 python35
```

Listing 1.1: Commands for macports to install Python 3 on Mac OS X.

How To Install SciPy

There are many ways to install SciPy. For example, two popular ways are to use package management on your platform (e.g. `dnf` on RedHat or macports on OS X) or use a Python package management tool, like `pip`. The SciPy documentation is excellent and covers how-to instructions for many different platforms on the official SciPy homepage:

- Install the SciPy Stack:
  https://www.scipy.org/install.html

When installing SciPy, ensure that you install the following packages as a minimum:

- SciPy
- NumPy
- Matplotlib
- Pandas
- Statsmodels

On Mac OS X with `macports`, you can type:

```
sudo port install py35-numpy py35-scipy py35-matplotlib py35-pandas py35-statsmodels
    py35-pip
sudo port select --set pip pip35
```

Listing 1.2: Commands for macports to install Python 3 SciPy libraries on Mac OS X.

On Fedora Linux with `dnf`, you can type:

```
sudo dnf install python3-numpy python3-scipy python3-pandas python3-matplotlib
    python3-statsmodels
```

Listing 1.3: Commands for `dnf` to install Python 3 SciPy libraries on Fedora Linux.

How To Install scikit-learn

The scikit-learn library must be installed separately. I would suggest that you use the same method to install scikit-learn as you used to install SciPy. There are instructions for installing scikit-learn, but they are limited to using the Python `pip` package manager. On all platforms that support `pip` (Windows, OS X and Linux), I installed scikit-learn by typing:

```
sudo pip install -U scikit-learn
```

Listing 1.4: Commands for `pip` to install scikit-learn.

1.3.3 Confirm Your Environment

Once you have setup your environment, you must confirm that it works as expected. Let's first check that Python was installed successfully. Open a command line and type:

```
python -V
```

Listing 1.5: Command line arguments to check Python version.

You should see a response like the following:

```
Python 2.7.12
```

Listing 1.6: Example Python version for Python 2.

or

```
Python 3.5.3
```

Listing 1.7: Example Python version for Python 3.

Now, confirm that the libraries were installed successfully. Create a new file called versions.py and copy and paste the following code snippet into it and save the file as `versions.py`.

```
# check the versions of key python libraries
# scipy
import scipy
print('scipy: %s' % scipy.__version__)
# numpy
import numpy
print('numpy: %s' % numpy.__version__)
# matplotlib
import matplotlib
print('matplotlib: %s' % matplotlib.__version__)
# pandas
import pandas
print('pandas: %s' % pandas.__version__)
# statsmodels
import statsmodels
print('statsmodels: %s' % statsmodels.__version__)
# scikit-learn
import sklearn
print('sklearn: %s' % sklearn.__version__)
```

Listing 1.8: Script to check versions of required Python libraries.

Run the file on the command line or in your favorite Python editor. For example, type:

```
python versions.py
```

Listing 1.9: Example of command line arguments to run the versions.py script.

This will print the version of each key library you require. For example, on my system at the time of writing, I got the following results:

```
scipy: 0.18.1
numpy: 1.12.0
matplotlib: 2.0.0
pandas: 0.19.2
statsmodels: 0.8.0
```

```
sklearn: 0.18.1
```

Listing 1.10: Example output of running the versions.py script.

If you have an error, stop now and fix it. You may need to consult the documentation specific for your platform.

1.4 Summary

In this lesson, you discovered the Python ecosystem for time series forecasting. You learned about:

- Pandas, Statsmodels, and scikit-learn that are the top Python libraries for time series forecasting.
- How to automatically and manually setup a Python SciPy environment for development.
- How to confirm your environment is installed correctly and that you're ready to started developing models.
- You also learned how to install the Python ecosystem for machine learning on your workstation.

1.4.1 Next

In the next lesson you will discover time series datasets and the standard terminology used when talking about time series problems.

Chapter 2

What is Time Series Forecasting?

Time series forecasting is an important area of machine learning that is often neglected. It is important because there are so many prediction problems that involve a time component. These problems are neglected because it is this time component that makes time series problems more difficult to handle. In this lesson, you will discover time series forecasting. After reading this lesson, you will know:

- Standard definitions of time series, time series analysis, and time series forecasting.
- The important components to consider in time series data.
- Examples of time series to make your understanding concrete.

Let's get started.

2.1 Time Series

A normal machine learning dataset is a collection of observations. For example:

```
observation #1
observation #2
observation #3
```

Listing 2.1: Example of a collection of observations.

Time does play a role in normal machine learning datasets. Predictions are made for new data when the actual outcome may not be known until some future date. The future is being predicted, but all prior observations are treated equally. Perhaps with some very minor temporal dynamics to overcome the idea of *concept drift* such as only using the last year of observations rather than all data available.

A time series dataset is different. Time series adds an explicit order dependence between observations: a time dimension. This additional dimension is both a constraint and a structure that provides a source of additional information.

> A time series is a sequence of observations taken sequentially in time.
>
> — Page 1, *Time Series Analysis: Forecasting and Control.*

For example:

```
Time #1, observation
Time #2, observation
Time #3, observation
```

Listing 2.2: Example of time series observations.

2.2 Time Series Nomenclature

Before we move on, it is important to quickly establish the standard terms used when describing time series data. The current time is defined as `t`, an observation at the current time is defined as `obs(t)`.

We are often interested in the observations made at prior times, called lag times or lags. Times in the past are negative relative to the current time. For example the previous time is `t-1` and the time before that is `t-2`. The observations at these times are `obs(t-1)` and `obs(t-2)` respectively. Times in the future are what we are interested in forecasting and are positive relative to the current time. For example the next time is `t+1` and the time after that is `t+2`. The observations at these times are `obs(t+1)` and `obs(t+2)` respectively.

For simplicity, we often drop the `obs(t)` notation and use `t+1` instead and assume we are talking about observations at times rather than the time indexes themselves. Additionally, we can refer to an observation at a lag by shorthand such as *a lag of 10* or `lag=10` which would be the same as `t-10`. To summarize:

- `t-n`: A prior or lag time (e.g. `t-1` for the previous time).
- `t`: A current time and point of reference.
- `t+n`: A future or forecast time (e.g. `t+1` for the next time).

2.3 Describing vs. Predicting

We have different goals depending on whether we are interested in understanding a dataset or making predictions. Understanding a dataset, called time series analysis, can help to make better predictions, but is not required and can result in a large technical investment in time and expertise not directly aligned with the desired outcome, which is forecasting the future.

> In descriptive modeling, or time series analysis, a time series is modeled to determine its components in terms of seasonal patterns, trends, relation to external factors, and the like. [...] In contrast, time series forecasting uses the information in a time series (perhaps with additional information) to forecast future values of that series
>
> — Page 18-19, *Practical Time Series Forecasting with R: A Hands-On Guide.*

2.3.1 Time Series Analysis

When using classical statistics, the primary concern is the analysis of time series. Time series analysis involves developing models that best capture or describe an observed time series in order to understand the underlying causes. This field of study seeks the *why* behind a time series dataset. This often involves making assumptions about the form of the data and decomposing the time series into constitution components. The quality of a descriptive model is determined by how well it describes all available data and the interpretation it provides to better inform the problem domain.

> The primary objective of time series analysis is to develop mathematical models that provide plausible descriptions from sample data
>
> — Page 11, *Time Series Analysis and Its Applications: With R Examples.*

2.3.2 Time Series Forecasting

Making predictions about the future is called extrapolation in the classical statistical handling of time series data. More modern fields focus on the topic and refer to it as time series forecasting. Forecasting involves taking models fit on historical data and using them to predict future observations. Descriptive models can borrow from the future (i.e. to smooth or remove noise), they only seek to best describe the data. An important distinction in forecasting is that the future is completely unavailable and must only be estimated from what has already happened.

The skill of a time series forecasting model is determined by its performance at predicting the future. This is often at the expense of being able to explain why a specific prediction was made, confidence intervals and even better understanding the underlying causes behind the problem.

2.4 Components of Time Series

Time series analysis provides a body of techniques to better understand a dataset. Perhaps the most useful of these is the decomposition of a time series into 4 constituent parts:

- **Level**. The baseline value for the series if it were a straight line.
- **Trend**. The optional and often linear increasing or decreasing behavior of the series over time.
- **Seasonality**. The optional repeating patterns or cycles of behavior over time.
- **Noise**. The optional variability in the observations that cannot be explained by the model.

All time series have a level, most have noise, and the trend and seasonality are optional.

> The main features of many time series are trends and seasonal variations [...] another important feature of most time series is that observations close together in time tend to be correlated (serially dependent)

— Page 2, *Introductory Time Series with R.*

These constituent components can be thought to combine in some way to provide the observed time series. Assumptions can be made about these components both in behavior and in how they are combined, which allows them to be modeled using traditional statistical methods. These components may also be the most effective way to make predictions about future values, but not always. In cases where these classical methods do not result in effective performance, these components may still be useful concepts, and even input to alternate methods. The decomposition of time series into level, trend, seasonality and noise is covered more in Chapter 12.

2.5 Concerns of Forecasting

When forecasting, it is important to understand your goal. Use the Socratic method and ask lots of questions to help zoom in on the specifics of your predictive modeling problem. For example:

1. **How much data do you have available and are you able to gather it all together?** More data is often more helpful, offering greater opportunity for exploratory data analysis, model testing and tuning, and model fidelity.
2. **What is the time horizon of predictions that is required?** Short, medium or long term? Shorter time horizons are often easier to predict with higher confidence.
3. **Can forecasts be updated frequently over time or must they be made once and remain static?** Updating forecasts as new information becomes available often results in more accurate predictions.
4. **At what temporal frequency are forecasts required?** Often forecasts can be made at a lower or higher frequencies, allowing you to harness down-sampling, and up-sampling of data, which in turn can offer benefits while modeling.

Time series data often requires cleaning, scaling, and even transformation. For example:

- **Frequency**. Perhaps data is provided at a frequency that is too high to model or is unevenly spaced through time requiring resampling for use in some models.
- **Outliers**. Perhaps there are corrupt or extreme outlier values that need to be identified and handled.
- **Missing**. Perhaps there are gaps or missing data that need to be interpolated or imputed.

Often time series problems are real-time, continually providing new opportunities for prediction. This adds an honesty to time series forecasting that quickly fleshes out bad assumptions, errors in modeling and all the other ways that we may be able to fool ourselves.

2.6 Examples of Time Series Forecasting

There is almost an endless supply of time series forecasting problems. Below are 10 examples from a range of industries to make the notions of time series analysis and forecasting more concrete.

- Forecasting the corn yield in tons by state each year.
- Forecasting whether an EEG trace in seconds indicates a patient is having a seizure or not.
- Forecasting the closing price of a stock each day.
- Forecasting the birth rate at all hospitals in a city each year.
- Forecasting product sales in units sold each day for a store.
- Forecasting the number of passengers through a train station each day.
- Forecasting unemployment for a state each quarter.
- Forecasting utilization demand on a server each hour.
- Forecasting the size of the rabbit population in a state each breeding season.
- Forecasting the average price of gasoline in a city each day.

I expect that you will be able to relate one or more of these examples to your own time series forecasting problems that you would like to address.

2.7 Summary

In this lesson, you discovered time series forecasting. Specifically, you learned:

- About time series data and the difference between time series analysis and time series forecasting.
- The constituent components that a time series may be decomposed into when performing an analysis.
- Examples of time series forecasting problems to make these ideas concrete.

2.7.1 Next

In the next lesson you will discover how to frame time series forecasting as a supervised learning problem.

Chapter 3

Time Series as Supervised Learning

Time series forecasting can be framed as a supervised learning problem. This re-framing of your time series data allows you access to the suite of standard linear and nonlinear machine learning algorithms on your problem. In this lesson, you will discover how you can re-frame your time series problem as a supervised learning problem for machine learning. After reading this lesson, you will know:

- What supervised learning is and how it is the foundation for all predictive modeling machine learning algorithms.
- The sliding window method for framing a time series dataset and how to use it.
- How to use the sliding window for multivariate data and multi-step forecasting.

Let's get started.

3.1 Supervised Machine Learning

The majority of practical machine learning uses supervised learning. Supervised learning is where you have input variables (`X`) and an output variable (`y`) and you use an algorithm to learn the mapping function from the input to the output.

$$Y = f(X) \tag{3.1}$$

The goal is to approximate the real underlying mapping so well that when you have new input data (`X`), you can predict the output variables (`y`) for that data. Below is a contrived example of a supervised learning dataset where each row is an observation comprised of one input variable (`X`) and one output variable to be predicted (`y`).

```
X,  y
5,  0.9
4,  0.8
5,  1.0
3,  0.7
4,  0.9
```

Listing 3.1: Example of a small contrived supervised learning dataset.

It is called supervised learning because the process of an algorithm learning from the training dataset can be thought of as a teacher supervising the learning process. We know the correct answers; the algorithm iteratively makes predictions on the training data and is corrected by making updates. Learning stops when the algorithm achieves an acceptable level of performance. Supervised learning problems can be further grouped into regression and classification problems.

- **Classification**: A classification problem is when the output variable is a category, such as `red` and `blue` or `disease` and `no disease`.
- **Regression**: A regression problem is when the output variable is a real value, such as `dollars` or `weight`. The contrived example above is a regression problem.

3.2 Sliding Window

Time series data can be phrased as supervised learning. Given a sequence of numbers for a time series dataset, we can restructure the data to look like a supervised learning problem. We can do this by using previous time steps as input variables and use the next time step as the output variable. Let's make this concrete with an example. Imagine we have a time series as follows:

```
time,  measure
1,     100
2,     110
3,     108
4,     115
5,     120
```

Listing 3.2: Example of a small contrived time series dataset.

We can restructure this time series dataset as a supervised learning problem by using the value at the previous time step to predict the value at the next time-step. Re-organizing the time series dataset this way, the data would look as follows:

```
X,   y
?,   100
100, 110
110, 108
108, 115
115, 120
120, ?
```

Listing 3.3: Example of time series dataset as supervised learning.

Take a look at the above transformed dataset and compare it to the original time series. Here are some observations:

- We can see that the previous time step is the input (`X`) and the next time step is the output (`y`) in our supervised learning problem.
- We can see that the order between the observations is preserved, and must continue to be preserved when using this dataset to train a supervised model.
- We can see that we have no previous value that we can use to predict the first value in the sequence. We will delete this row as we cannot use it.

- We can also see that we do not have a known next value to predict for the last value in the sequence. We may want to delete this value while training our supervised model also.

The use of prior time steps to predict the next time step is called the sliding window method. For short, it may be called the window method in some literature. In statistics and time series analysis, this is called a lag or lag method. The number of previous time steps is called the window width or size of the lag. This sliding window is the basis for how we can turn any time series dataset into a supervised learning problem. From this simple example, we can notice a few things:

- We can see how this can work to turn a time series into either a regression or a classification supervised learning problem for real-valued or labeled time series values.
- We can see how once a time series dataset is prepared this way that any of the standard linear and nonlinear machine learning algorithms may be applied, as long as the order of the rows is preserved.
- We can see how the width sliding window can be increased to include more previous time steps.
- We can see how the sliding window approach can be used on a time series that has more than one value, or so-called multivariate time series.

We will explore some of these uses of the sliding window, starting next with using it to handle time series with more than one observation at each time step, called multivariate time series.

3.3 Sliding Window With Multivariates

The number of observations recorded for a given time in a time series dataset matters. Traditionally, different names are used:

- **Univariate Time Series**: These are datasets where only a single variable is observed at each time, such as temperature each hour. The example in the previous section is a univariate time series dataset.
- **Multivariate Time Series**: These are datasets where two or more variables are observed at each time.

Most time series analysis methods, and even books on the topic, focus on univariate data. This is because it is the simplest to understand and work with. Multivariate data is often more difficult to work with. It is harder to model and often many of the classical methods do not perform well.

> Multivariate time series analysis considers simultaneously multiple time series. [...]
> It is, in general, much more complicated than univariate time series analysis
>
> — Page 1, *Multivariate Time Series Analysis: With R and Financial Applications.*

The sweet spot for using machine learning for time series is where classical methods fall down. This may be with complex univariate time series, and is more likely with multivariate time series given the additional complexity. Below is another worked example to make the sliding window method concrete for multivariate time series. Assume we have the contrived multivariate time series dataset below with two observations at each time step. Let's also assume that we are only concerned with predicting `measure2`.

```
time,  measure1,  measure2
1,     0.2,       88
2,     0.5,       89
3,     0.7,       87
4,     0.4,       88
5,     1.0,       90
```

Listing 3.4: Example of a small contrived multivariate time series dataset.

We can re-frame this time series dataset as a supervised learning problem with a window width of one. This means that we will use the previous time step values of `measure1` and `measure2`. We will also have available the next time step value for `measure1`. We will then predict the next time step value of `measure2`. This will give us 3 input features and one output value to predict for each training pattern.

```
X1,   X2,   X3,   y
?,    ?,    0.2,  88
0.2,  88,   0.5,  89
0.5,  89,   0.7,  87
0.7,  87,   0.4,  88
0.4,  88,   1.0,  90
1.0,  90,   ?,    ?
```

Listing 3.5: Example of a multivariate time series dataset as a supervised learning problem.

We can see that as in the univariate time series example above, we may need to remove the first and last rows in order to train our supervised learning model. This example raises the question of what if we wanted to predict both `measure1` and `measure2` for the next time step? The sliding window approach can also be used in this case. Using the same time series dataset above, we can phrase it as a supervised learning problem where we predict both `measure1` and `measure2` with the same window width of one, as follows.

```
X1,   X2,   y1,   y2
?,    ?,    0.2,  88
0.2,  88,   0.5,  89
0.5,  89,   0.7,  87
0.7,  87,   0.4,  88
0.4,  88,   1.0,  90
1.0,  90,   ?,    ?
```

Listing 3.6: Example of a multivariate time series dataset as a multi-step or sequence prediction supervised learning problem.

Not many supervised learning methods can handle the prediction of multiple output values without modification, but some methods, like artificial neural networks, have little trouble. We can think of predicting more than one value as predicting a sequence. In this case, we were predicting two different output variables, but we may want to predict multiple time-steps ahead of one output variable. This is called multi-step forecasting and is covered in the next section.

3.4 Sliding Window With Multiple Steps

The number of time steps ahead to be forecasted is important. Again, it is traditional to use different names for the problem depending on the number of time-steps to forecast:

- **One-Step Forecast**: This is where the next time step (`t+1`) is predicted.
- **Multi-Step Forecast**: This is where two or more future time steps are to be predicted.

All of the examples we have looked at so far have been one-step forecasts. There are a number of ways to model multi-step forecasting as a supervised learning problem. For now, we are focusing on framing multi-step forecast using the sliding window method. Consider the same univariate time series dataset from the first sliding window example above:

```
time, measure
1,    100
2,    110
3,    108
4,    115
5,    120
```

Listing 3.7: Example of a small contrived time series dataset.

We can frame this time series as a two-step forecasting dataset for supervised learning with a window width of one, as follows:

```
X1,   y1,   y2
?     100,  110
100,  110,  108
110,  108,  115
108,  115,  120
115,  120,  ?
120,  ?,    ?
```

Listing 3.8: Example of a univariate time series dataset as a multi-step or sequence prediction supervised learning problem.

We can see that the first row and the last two rows cannot be used to train a supervised model. It is also a good example to show the burden on the input variables. Specifically, that a supervised model only has `X1` to work with in order to predict both `y1` and `y2`. Careful thought and experimentation are needed on your problem to find a window width that results in acceptable model performance.

3.5 Summary

In this lesson, you discovered how you can re-frame your time series prediction problem as a supervised learning problem for use with machine learning methods. Specifically, you learned:

- Supervised learning is the most popular way of framing problems for machine learning as a collection of observations with inputs and outputs.
- Sliding window is the way to restructure a time series dataset as a supervised learning problem.

- Multivariate and multi-step forecasting time series can also be framed as supervised learning using the sliding window method.

3.5.1 Next

This concludes Part I of the book. Next in Part II you will learn about data preparation for time series, starting with how to load time series data.

Part II

Data Preparation

Chapter 4

Load and Explore Time Series Data

The Pandas library in Python provides excellent, built-in support for time series data. Once loaded, Pandas also provides tools to explore and better understand your dataset. In this lesson, you will discover how to load and explore your time series dataset. After completing this tutorial, you will know:

- How to load your time series dataset from a CSV file using Pandas.
- How to peek at the loaded data and query using date-times.
- How to calculate and review summary statistics.

Let's get started.

4.1 Daily Female Births Dataset

In this lesson, we will use the Daily Female Births Dataset as an example. This dataset describes the number of daily female births in California in 1959. You can learn more about the dataset in Appendix A.4. Place the dataset in your current working directory with the filename `daily-total-female-births.csv`.

4.2 Load Time Series Data

Pandas represented time series datasets as a Series. A `Series`[1] is a one-dimensional array with a time label for each row. We can load the Daily Female Births dataset directly using the `Series` class as follows:

```
# load dataset using Series.from_csv()
from pandas import Series
series = Series.from_csv('daily-total-female-births.csv', header=0)
print(series.head())
```

Listing 4.1: Example of loading a CSV using the Pandas `Series`.

Running this example prints the first 5 rows of the dataset, as follows:

[1] http://pandas.pydata.org/pandas-docs/stable/generated/pandas.Series.html

```
Date
1959-01-01   35
1959-01-02   32
1959-01-03   30
1959-01-04   31
1959-01-05   44
Name: Births, dtype: int64
```

Listing 4.2: Output of loading the dataset as a Pandas `Series`.

The series has a name, which is the column name of the data column. You can see that each row has an associated date. This is in fact not a column, but instead a time index for value. As an index, there can be multiple values for one time, and values may be spaced evenly or unevenly across times. The main function for loading CSV data in Pandas is the `read_csv()` function[2]. We can use this to load the time series as a `Series` object, instead of a `DataFrame`, as follows:

```
# load dataset using read_csv()
from pandas import read_csv
series = read_csv('daily-total-female-births.csv', header=0, parse_dates=[0], index_col=0,
    squeeze=True)
print(type(series))
print(series.head())
```

Listing 4.3: Example of loading a CSV using `read_csv()`.

Note the arguments to the `read_csv()` function. We provide it a number of hints to ensure the data is loaded as a `Series`.

- `header=0`: We must specify the header information at row 0.
- `parse_dates=[0]`: We give the function a hint that data in the first column contains dates that need to be parsed. This argument takes a list, so we provide it a list of one element, which is the index of the first column.
- `index_col=0`: We hint that the first column contains the index information for the time series.
- `squeeze=True`: We hint that we only have one data column and that we are interested in a `Series` and not a `DataFrame`.

One more argument you may need to use for your own data is `date_parser` to specify the function to parse date-time values. In this example, the date format has been inferred, and this works in most cases. In those few cases where it does not, specify your own date parsing function and use the `date_parser` argument. Running the example above prints the same output, but also confirms that the time series was indeed loaded as a `Series` object.

```
<class 'pandas.core.series.Series'>
Date
1959-01-01   35
```

[2] http://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html#pandas.read_csv

```
1959-01-02   32
1959-01-03   30
1959-01-04   31
1959-01-05   44
Name: Births, dtype: int64
```

Listing 4.4: Output of loading a CSV using `read_csv()`.

4.3 Exploring Time Series Data

Pandas also provides tools to explore and summarize your time series data. In this section, we'll take a look at a few, common operations to explore and summarize your loaded time series data.

4.3.1 Peek at the Data

It is a good idea to take a peek at your loaded data to confirm that the types, dates, and data loaded as you intended. You can use the `head()` function to peek at the first 5 records or specify the first `n` number of records to review. For example, you can print the first 10 rows of data as follows.

```
# summarize first few lines of a file
from pandas import Series
series = Series.from_csv('daily-total-female-births.csv', header=0)
print(series.head(10))
```

Listing 4.5: Example of printing the first 10 rows of a time series.

Running the example prints the following:

```
Date
1959-01-01   35
1959-01-02   32
1959-01-03   30
1959-01-04   31
1959-01-05   44
1959-01-06   29
1959-01-07   45
1959-01-08   43
1959-01-09   38
1959-01-10   27
Name: Births, dtype: int64
```

Listing 4.6: Output of printing the first 10 rows of a dataset.

You can also use the `tail()` function to get the last `n` records of the dataset.

4.3.2 Number of Observations

Another quick check to perform on your data is the number of loaded observations. This can help flush out issues with column headers not being handled as intended, and to get an idea on how to effectively divide up data later for use with supervised learning algorithms. You can get the dimensionality of your `Series` using the `size` parameter.

```
# summarize the dimensions of a time series
from pandas import Series
series = Series.from_csv('daily-total-female-births.csv', header=0)
print(series.size)
```

Listing 4.7: Example of printing the dimensions of a time series.

Running this example we can see that as we would expect, there are 365 observations, one for each day of the year in 1959.

```
365
```

Listing 4.8: Output of printing the dimensions of a dataset.

4.3.3 Querying By Time

You can slice, dice, and query your series using the time index. For example, you can access all observations in January as follows:

```
# query a dataset using a date-time index
from pandas import Series
series = Series.from_csv('daily-total-female-births.csv', header=0)
print(series['1959-01'])
```

Listing 4.9: Example of querying a time series by date-time.

Running this displays the 31 observations for the month of January in 1959.

```
Date
1959-01-01   35
1959-01-02   32
1959-01-03   30
1959-01-04   31
1959-01-05   44
1959-01-06   29
1959-01-07   45
1959-01-08   43
1959-01-09   38
1959-01-10   27
1959-01-11   38
1959-01-12   33
1959-01-13   55
1959-01-14   47
1959-01-15   45
1959-01-16   37
1959-01-17   50
1959-01-18   43
1959-01-19   41
1959-01-20   52
1959-01-21   34
1959-01-22   53
1959-01-23   39
1959-01-24   32
1959-01-25   37
1959-01-26   43
1959-01-27   39
```

```
1959-01-28   35
1959-01-29   44
1959-01-30   38
1959-01-31   24
Name: Births, dtype: int64
```

Listing 4.10: Output of querying a time series dataset.

This type of index-based querying can help to prepare summary statistics and plots while exploring the dataset.

4.3.4 Descriptive Statistics

Calculating descriptive statistics on your time series can help get an idea of the distribution and spread of values. This may help with ideas of data scaling and even data cleaning that you can perform later as part of preparing your dataset for modeling. The `describe()` function creates a 7 number summary of the loaded time series including mean, standard deviation, median, minimum, and maximum of the observations.

```
# calculate descriptive statistics
from pandas import Series
series = Series.from_csv('daily-total-female-births.csv', header=0)
print(series.describe())
```

Listing 4.11: Example of printing the descriptive stats of a time series.

Running this example prints a summary of the birth rate dataset.

```
count   365.000000
mean     41.980822
std       7.348257
min      23.000000
25%      37.000000
50%      42.000000
75%      46.000000
max      73.000000
Name: Births, dtype: float64
```

Listing 4.12: Output of printing the descriptive stats of a times series.

4.4 Summary

In this lesson, you discovered how to load and handle time series data using the Pandas Python library. Specifically, you learned:

- How to load your time series data as a Pandas `Series`.
- How to peek at your time series data and query it by date-time.
- How to calculate and review summary statistics on time series data.

4.4.1 Next

In the next lesson you will discover how to perform feature engineering with time series data.

Chapter 5

Basic Feature Engineering

Time Series data must be re-framed as a supervised learning dataset before we can start using machine learning algorithms. There is no concept of input and output features in time series. Instead, we must choose the variable to be predicted and use feature engineering to construct all of the inputs that will be used to make predictions for future time steps. In this tutorial, you will discover how to perform feature engineering on time series data with Python to model your time series problem with machine learning algorithms.

After completing this tutorial, you will know:

- The rationale and goals of feature engineering time series data.
- How to develop basic date-time based input features.
- How to develop more sophisticated lag and sliding window summary statistics features.

Let's dive in.

5.1 Feature Engineering for Time Series

A time series dataset must be transformed to be modeled as a supervised learning problem. That is something that looks like:

```
time 1, value 1
time 2, value 2
time 3, value 3
```

Listing 5.1: Example of a contrived time series.

To something that looks like:

```
input 1, output 1
input 2, output 2
input 3, output 3
```

Listing 5.2: Example of a contrived time series transformed to a supervised learning problem.

So that we can train a supervised learning algorithm. Input variables are also called features in the field of machine learning, and the task before us is to create or invent new input features from our time series dataset. Ideally, we only want input features that best help the learning

methods model the relationship between the inputs (`X`) and the outputs (`y`) that we would like to predict. In this tutorial, we will look at three classes of features that we can create from our time series dataset:

- **Date Time Features**: these are components of the time step itself for each observation.
- **Lag Features**: these are values at prior time steps.
- **Window Features**: these are a summary of values over a fixed window of prior time steps.

Before we dive into methods for creating input features from our time series data, let's first review the goal of feature engineering.

5.2 Goal of Feature Engineering

The goal of feature engineering is to provide strong and ideally simple relationships between new input features and the output feature for the supervised learning algorithm to model. In effect, we are moving complexity.

Complexity exists in the relationships between the input and output data. In the case of time series, there is no concept of input and output variables; we must invent these too and frame the supervised learning problem from scratch. We may lean on the capability of sophisticated models to decipher the complexity of the problem. We can make the job for these models easier (and even use simpler models) if we can better expose the inherent relationship between inputs and outputs in the data.

The difficulty is that we do not know the underlying inherent functional relationship between inputs and outputs that we're trying to expose. If we did know, we probably would not need machine learning. Instead, the only feedback we have is the performance of models developed on the supervised learning datasets or *views* of the problem we create. In effect, the best default strategy is to use all the knowledge available to create many good datasets from your time series dataset and use model performance (and other project requirements) to help determine what good features and good views of your problem happen to be.

For clarity, we will focus on a univariate (one variable) time series dataset in the examples, but these methods are just as applicable to multivariate time series problems. Next, let's take a look at the dataset we will use in this tutorial.

5.3 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`.

5.4 Date Time Features

Let's start with some of the simplest features that we can use. These are features from the date/time of each observation. In fact, these can start off simply and head off into quite complex domain-specific areas. Two features that we can start with are the integer month and day for each observation. We can imagine that supervised learning algorithms may be able to use these inputs to help tease out time-of-year or time-of-month type seasonality information. The supervised learning problem we are proposing is to predict the daily minimum temperature given the month and day, as follows:

```
Month, Day, Temperature
Month, Day, Temperature
Month, Day, Temperature
```

Listing 5.3: Example of month and day features.

We can do this using Pandas. First, the time series is loaded as a Pandas `Series`. We then create a new Pandas `DataFrame` for the transformed dataset. Next, each column is added one at a time where month and day information is extracted from the time-stamp information for each observation in the series. Below is the Python code to do this.

```
# create date time features of a dataset
from pandas import Series
from pandas import DataFrame
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
dataframe = DataFrame()
dataframe['month'] = [series.index[i].month for i in range(len(series))]
dataframe['day'] = [series.index[i].day for i in range(len(series))]
dataframe['temperature'] = [series[i] for i in range(len(series))]
print(dataframe.head(5))
```

Listing 5.4: Example of date-time features in the Minimum Daily Temperatures dataset.

Running this example prints the first 5 rows of the transformed dataset.

```
   month  day  temperature
0      1    1         20.7
1      1    2         17.9
2      1    3         18.8
3      1    4         14.6
4      1    5         15.8
```

Listing 5.5: Example output of date-time features in the Minimum Daily Temperatures dataset.

Using just the month and day information alone to predict temperature is not sophisticated and will likely result in a poor model. Nevertheless, this information coupled with additional engineered features may ultimately result in a better model. You may enumerate all the properties of a time-stamp and consider what might be useful for your problem, such as:

- Minutes elapsed for the day.
- Hour of day.
- Business hours or not.
- Weekend or not.

- Season of the year.
- Business quarter of the year.
- Daylight savings or not.
- Public holiday or not.
- Leap year or not.

From these examples, you can see that you're not restricted to the raw integer values. You can use binary flag features as well, like whether or not the observation was recorded on a public holiday. In the case of the minimum temperature dataset, maybe the season would be more relevant. It is creating domain-specific features like this that are more likely to add value to your model. Date-time based features are a good start, but it is often a lot more useful to include the values at previous time steps. These are called lagged values and we will look at adding these features in the next section.

5.5 Lag Features

Lag features are the classical way that time series forecasting problems are transformed into supervised learning problems. The simplest approach is to predict the value at the next time (`t+1`) given the value at the current time (`t`). The supervised learning problem with shifted values looks as follows:

```
Value(t), Value(t+1)
Value(t), Value(t+1)
Value(t), Value(t+1)
```

Listing 5.6: Example of lag features.

The Pandas library provides the `shift()` function[1] to help create these shifted or lag features from a time series dataset. Shifting the dataset by 1 creates the `t` column, adding a `NaN` (unknown) value for the first row. The time series dataset without a shift represents the `t+1`. Let's make this concrete with an example. The first 3 values of the temperature dataset are 20.7, 17.9, and 18.8. The shifted and unshifted lists of temperatures for the first 3 observations are therefore:

```
Shifted, Original
NaN,     20.7
20.7,    17.9
17.9,    18.8
```

Listing 5.7: Example of manually shifted rows.

We can concatenate the shifted columns together into a new `DataFrame` using the `concat()` function[2] along the column axis (`axis=1`). Putting this all together, below is an example of creating a lag feature for our daily temperature dataset. The values are extracted from the loaded series and a shifted and unshifted list of these values is created. Each column is also named in the `DataFrame` for clarity.

[1] http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.shift.html
[2] http://pandas.pydata.org/pandas-docs/stable/generated/pandas.concat.html

```
# create a lag feature
from pandas import Series
from pandas import DataFrame
from pandas import concat
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
temps = DataFrame(series.values)
dataframe = concat([temps.shift(1), temps], axis=1)
dataframe.columns = ['t', 't+1']
print(dataframe.head(5))
```

Listing 5.8: Example of `lag=1` features on the Minimum Daily Temperatures dataset.

Running the example prints the first 5 rows of the new dataset with the lagged feature.

```
    t     t+1
0   NaN  20.7
1  20.7  17.9
2  17.9  18.8
3  18.8  14.6
4  14.6  15.8
```

Listing 5.9: Example output of `lag=1` features.

You can see that we would have to discard the first row to use the dataset to train a supervised learning model, as it does not contain enough data to work with. The addition of lag features is called the sliding window method, in this case with a window width of 1. It is as though we are sliding our focus along the time series for each observation with an interest in only what is within the window width. We can expand the window width and include more lagged features. For example, below is the above case modified to include the last 3 observed values to predict the value at the next time step.

```
# create lag features
from pandas import Series
from pandas import DataFrame
from pandas import concat
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
temps = DataFrame(series.values)
dataframe = concat([temps.shift(3), temps.shift(2), temps.shift(1), temps], axis=1)
dataframe.columns = ['t-2', 't-1', 't', 't+1']
print(dataframe.head(5))
```

Listing 5.10: Example of lag

Running this example prints the first 5 rows of the new lagged dataset.

```
    t-2   t-1   t     t+1
0   NaN   NaN   NaN  20.7
1   NaN   NaN  20.7  17.9
2   NaN  20.7  17.9  18.8
3  20.7  17.9  18.8  14.6
4  17.9  18.8  14.6  15.8
```

Listing 5.11: Example output of lag=3 features.

Again, you can see that we must discard the first few rows that do not have enough data to train a supervised model. A difficulty with the sliding window approach is how large to make

the window for your problem. Perhaps a good starting point is to perform a sensitivity analysis and try a suite of different window widths to in turn create a suite of different *views* of your dataset and see which results in better performing models. There will be a point of diminishing returns.

Additionally, why stop with a linear window? Perhaps you need a lag value from last week, last month, and last year. Again, this comes down to the specific domain. In the case of the temperature dataset, a lag value from the same day in the previous year or previous few years may be useful. We can do more with a window than include the raw values. In the next section, we'll look at including features that summarize statistics across the window.

5.6 Rolling Window Statistics

A step beyond adding raw lagged values is to add a summary of the values at previous time steps. We can calculate summary statistics across the values in the sliding window and include these as features in our dataset. Perhaps the most useful is the mean of the previous few values, also called the rolling mean.

We can calculate the mean of the current and previous values and use that to predict the next value. For the temperature data, we would have to wait 3 time steps before we had 2 values to take the average of before we could use that value to predict a 3rd value. For example:

```
mean(t-1, t), t+1
mean(20.7, 17.9), 18.8
19.3, 18.8
```

Listing 5.12: Example of manual rolling mean features.

Pandas provides a `rolling()` function[3] that creates a new data structure with the window of values at each time step. We can then perform statistical functions on the window of values collected for each time step, such as calculating the mean. First, the series must be shifted. Then the rolling dataset can be created and the mean values calculated on each window of two values. Here are the values in the first three rolling windows:

```
#,  Window Values
1,  NaN
2,  NaN, 20.7
3,  20.7, 17.9
```

Listing 5.13: Example of manual rolling window features.

This suggests that we will not have usable data until the 3rd row. Finally, as in the previous section, we can use the `concat()` function to construct a new dataset with just our new columns. The example below demonstrates how to do this with Pandas with a window size of 2.

```
# create a rolling mean feature
from pandas import Series
from pandas import DataFrame
from pandas import concat
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
temps = DataFrame(series.values)
shifted = temps.shift(1)
```

[3] http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.rolling.html

```
window = shifted.rolling(window=2)
means = window.mean()
dataframe = concat([means, temps], axis=1)
dataframe.columns = ['mean(t-1,t)', 't+1']
print(dataframe.head(5))
```

Listing 5.14: Example of rolling mean feature on the Minimum Daily Temperatures dataset.

Running the example prints the first 5 rows of the new dataset. We can see that the first two rows are not useful.

- The first `NaN` was created by the shift of the series.
- The second because `NaN` cannot be used to calculate a mean value.
- Finally, the third row shows the expected value of 19.30 (the mean of 20.7 and 17.9) used to predict the 3rd value in the series of 18.8.

```
   mean(t-1,t)   t+1
0          NaN  20.7
1          NaN  17.9
2        19.30  18.8
3        18.35  14.6
4        16.70  15.8
```

Listing 5.15: Example output of rolling mean feature on the Minimum Daily Temperatures dataset.

There are more statistics we can calculate and even different mathematical ways of calculating the definition of the *window*. Below is another example that shows a window width of 3 and a dataset comprised of more summary statistics, specifically the minimum, mean, and maximum value in the window.

You can see in the code that we are explicitly specifying the sliding window width as a named variable. This allows us to use it both in calculating the correct shift of the series and in specifying the width of the window to the `rolling()` function.

In this case, the window width of 3 means we must shift the series forward by 2 time steps. This makes the first two rows `NaN`. Next, we need to calculate the window statistics with 3 values per window. It takes 3 rows before we even have enough data from the series in the window to start calculating statistics. The values in the first 5 windows are as follows:

```
#, Window Values
1, NaN
2, NaN, NaN
3, NaN, NaN, 20.7
4, NaN, 20.7, 17.9
5, 20.7, 17.9, 18.8
```

Listing 5.16: Example of manual rolling mean features.

This suggests that we would not expect usable data until at least the 5th row (array index 4)

```
# create rolling statistics features
from pandas import Series
from pandas import DataFrame
```

```
from pandas import concat
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
temps = DataFrame(series.values)
width = 3
shifted = temps.shift(width - 1)
window = shifted.rolling(window=width)
dataframe = concat([window.min(), window.mean(), window.max(), temps], axis=1)
dataframe.columns = ['min', 'mean', 'max', 't+1']
print(dataframe.head(5))
```

Listing 5.17: Example of rolling stats features on the Minimum Daily Temperatures dataset.

Running the code prints the first 5 rows of the new dataset. We can spot-check the correctness of the values on the 5th row (array index 4). We can see that indeed 17.9 is the minimum and 20.7 is the maximum of values in the window of `[20.7, 17.9, 18.8]`.

```
    min       mean   max   t+1
0   NaN        NaN   NaN  20.7
1   NaN        NaN   NaN  17.9
2   NaN        NaN   NaN  18.8
3   NaN        NaN   NaN  14.6
4  17.9  19.133333  20.7  15.8
```

Listing 5.18: Example output of rolling stats features on the Minimum Daily Temperatures dataset.

5.7 Expanding Window Statistics

Another type of window that may be useful includes all previous data in the series. This is called an expanding window and can help with keeping track of the bounds of observable data. Like the `rolling()` function on `DataFrame`, Pandas provides an `expanding()` function[4] that collects sets of all prior values for each time step.

These lists of prior numbers can be summarized and included as new features. For example, below are the lists of numbers in the expanding window for the first 5 time steps of the series:

```
#, Window Values
1, 20.7
2, 20.7, 17.9,
3, 20.7, 17.9, 18.8
4, 20.7, 17.9, 18.8, 14.6
5, 20.7, 17.9, 18.8, 14.6, 15.8
```

Listing 5.19: Example of manual expanding window.

Again, you can see that we must shift the series one-time step to ensure that the output value we wish to predict is excluded from these window values. Therefore the input windows look as follows:

```
#, Window Values
1, NaN
2, NaN, 20.7
3, NaN, 20.7, 17.9,
```

[4] http://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.expanding.html

```
4, NaN, 20.7, 17.9, 18.8
5, NaN, 20.7, 17.9, 18.8, 14.6
```

Listing 5.20: Example of manual expanding window with shift.

Thankfully, the statistical calculations exclude the `NaN` values in the expanding window, meaning no further modification is required. Below is an example of calculating the minimum, mean, and maximum values of the expanding window on the daily temperature dataset.

```
# create expanding window features
from pandas import Series
from pandas import DataFrame
from pandas import concat
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
temps = DataFrame(series.values)
window = temps.expanding()
dataframe = concat([window.min(), window.mean(), window.max(), temps.shift(-1)], axis=1)
dataframe.columns = ['min', 'mean', 'max', 't+1']
print(dataframe.head(5))
```

Listing 5.21: Example expanding stats features on the Minimum Daily Temperatures dataset.

Running the example prints the first 5 rows of the dataset. Spot-checking the expanding minimum, mean, and maximum values shows the example having the intended effect.

```
    min       mean   max   t+1
0  20.7  20.700000 20.7  17.9
1  17.9  19.300000 20.7  18.8
2  17.9  19.133333 20.7  14.6
3  14.6  18.000000 20.7  15.8
4  14.6  17.560000 20.7  15.8
```

Listing 5.22: Example output of expanding stats features on the Minimum Daily Temperatures dataset.

5.8 Summary

In this tutorial, you discovered how to use feature engineering to transform a time series dataset into a supervised learning dataset for machine learning. Specifically, you learned:

- The importance and goals of feature engineering time series data.
- How to develop date-time and lag-based features.
- How to develop sliding and expanding window summary statistic features.

5.8.1 Next

In the next lesson you will discover how to visualize time series data with a suite of different plots.

Chapter 6

Data Visualization

Time series lends itself naturally to visualization. Line plots of observations over time are popular, but there is a suite of other plots that you can use to learn more about your problem. The more you learn about your data, the more likely you are to develop a better forecasting model. In this tutorial, you will discover 6 different types of plots that you can use to visualize time series data with Python. Specifically, after completing this tutorial, you will know:

- How to explore the temporal structure of time series with line plots, lag plots, and autocorrelation plots.
- How to understand the distribution of observations using histograms and density plots.
- How to tease out the change in distribution over intervals using box and whisker plots and heat map plots.

Let's get started.

6.1 Time Series Visualization

Visualization plays an important role in time series analysis and forecasting. Plots of the raw sample data can provide valuable diagnostics to identify temporal structures like trends, cycles, and seasonality that can influence the choice of model. A problem is that many novices in the field of time series forecasting stop with line plots. In this tutorial, we will take a look at 6 different types of visualizations that you can use on your own time series data. They are:

1. Line Plots.
2. Histograms and Density Plots.
3. Box and Whisker Plots.
4. Heat Maps.
5. Lag Plots or Scatter Plots.
6. Autocorrelation Plots.

The focus is on univariate time series, but the techniques are just as applicable to multivariate time series, when you have more than one observation at each time step. Next, let's take a look at the dataset we will use to demonstrate time series visualization in this tutorial.

6.2 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`.

6.3 Line Plot

The first, and perhaps most popular, visualization for time series is the line plot. In this plot, time is shown on the x-axis with observation values along the y-axis. Below is an example of visualizing the Pandas `Series` of the Minimum Daily Temperatures dataset directly as a line plot.

```
# create a line plot
from pandas import Series
from matplotlib import pyplot
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
series.plot()
pyplot.show()
```

Listing 6.1: Example a Line Plot on the Minimum Daily Temperatures dataset.

Running the example creates a line plot.

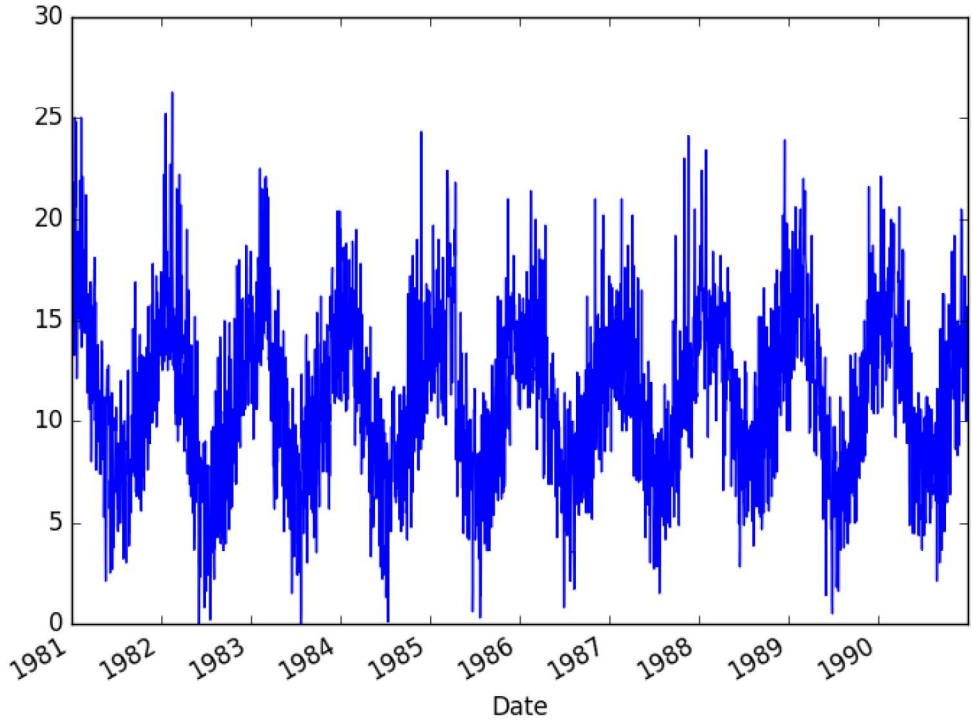

Figure 6.1: Line plot of the Minimum Daily Temperatures dataset.

The line plot is quite dense. Sometimes it can help to change the style of the line plot; for example, to use a dashed line or dots. Below is an example of changing the style of the line to be black dots instead of a connected line (the `style='k.'` argument). We could change this example to use a dashed line by setting style to be `'k--'`.

```
# create a dot plot
from pandas import Series
from matplotlib import pyplot
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
series.plot(style='k.')
pyplot.show()
```

Listing 6.2: Example a Dot Line Plot on the Minimum Daily Temperatures dataset.

Running the example recreates the same line plot with dots instead of the connected line.

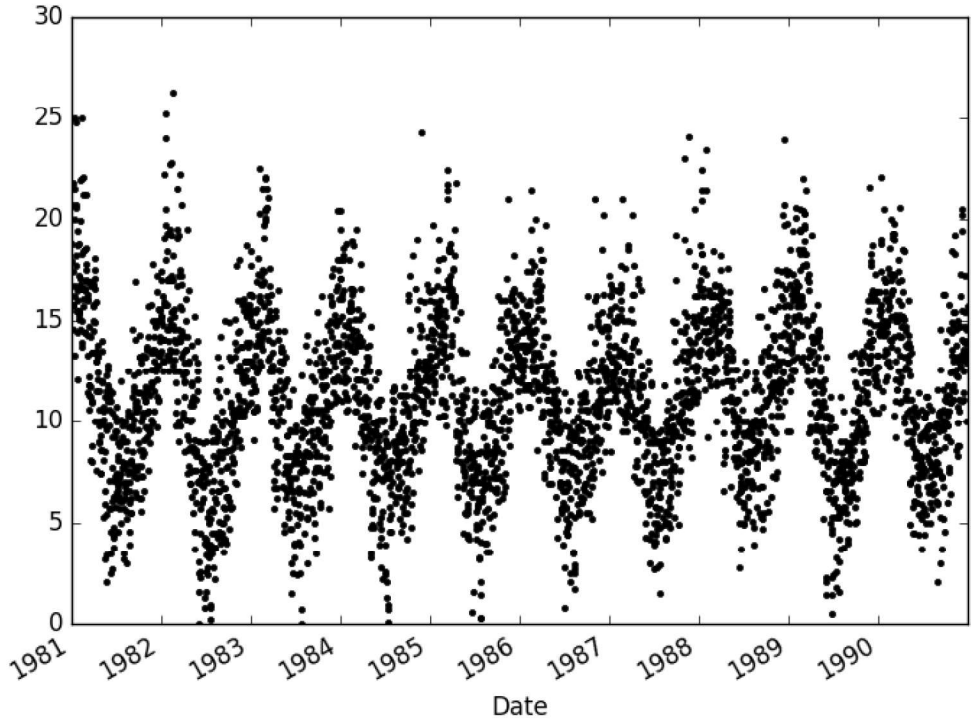

Figure 6.2: Dot line plot of the Minimum Daily Temperatures dataset.

It can be helpful to compare line plots for the same interval, such as from day-to-day, month-to-month, and year-to-year. The Minimum Daily Temperatures dataset spans 10 years. We can group data by year and create a line plot for each year for direct comparison. The example below shows how to do this. First the observations are grouped by year (`series.groupby(TimeGrouper('A'))`).

The groups are then enumerated and the observations for each year are stored as columns in a new `DataFrame`. Finally, a plot of this contrived `DataFrame` is created with each column visualized as a subplot with legends removed to cut back on the clutter.

```
# create stacked line plots
from pandas import Series
from pandas import DataFrame
from pandas import TimeGrouper
from matplotlib import pyplot
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
groups = series.groupby(TimeGrouper('A'))
years = DataFrame()
for name, group in groups:
  years[name.year] = group.values
years.plot(subplots=True, legend=False)
pyplot.show()
```

Listing 6.3: Example a Stacked Line Plots on the Minimum Daily Temperatures dataset.

Running the example creates 10 line plots, one for each year from 1981 at the top and 1990 at the bottom, where each line plot is 365 days in length.

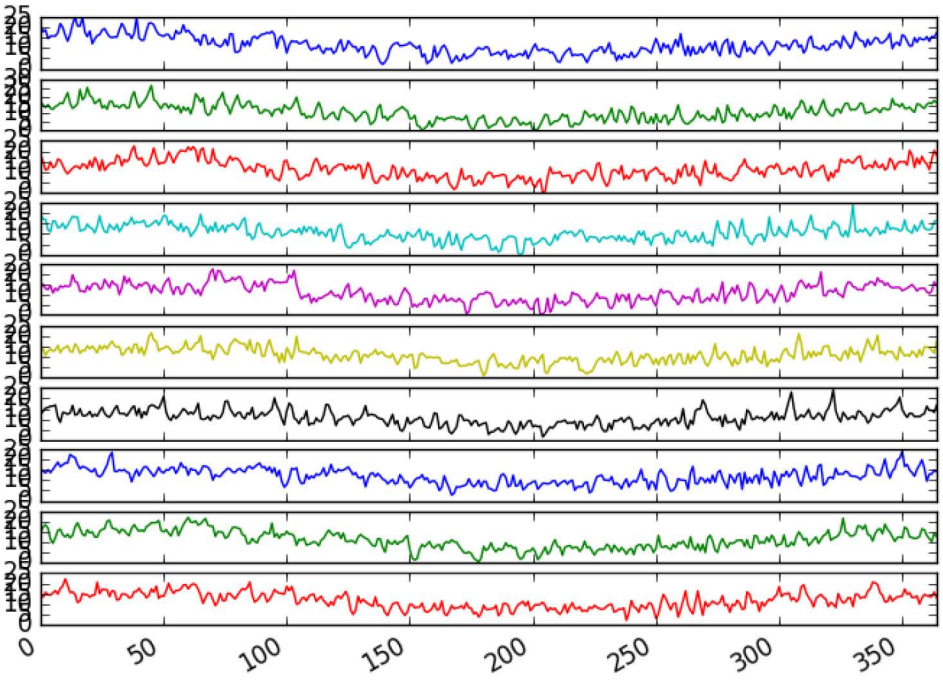

Figure 6.3: Stacked line plots of the Minimum Daily Temperatures dataset.

6.4 Histogram and Density Plots

Another important visualization is of the distribution of observations themselves. This means a plot of the values without the temporal ordering. Some linear time series forecasting methods assume a well-behaved distribution of observations (i.e. a bell curve or normal distribution). This can be explicitly checked using tools like statistical hypothesis tests. But plots can provide a useful first check of the distribution of observations both on raw observations and after any type of data transform has been performed.

The example below creates a histogram plot of the observations in the Minimum Daily Temperatures dataset. A histogram groups values into bins, and the frequency or count of observations in each bin can provide insight into the underlying distribution of the observations.

```
# create a histogram plot
from pandas import Series
from matplotlib import pyplot
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
series.hist()
```

```
pyplot.show()
```

Listing 6.4: Example a Histogram on the Minimum Daily Temperatures dataset.

Running the example shows a distribution that looks strongly Gaussian. The plotting function automatically selects the size of the bins based on the spread of values in the data.

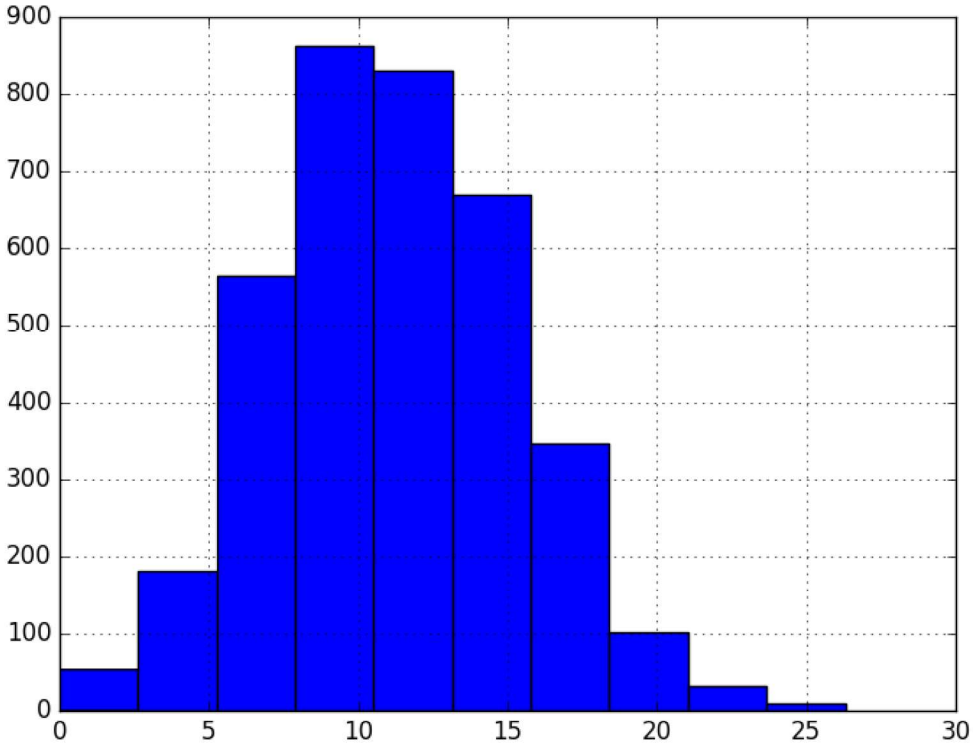

Figure 6.4: Histogram of the Minimum Daily Temperatures dataset.

We can get a better idea of the shape of the distribution of observations by using a density plot. This is like the histogram, except a function is used to fit the distribution of observations and a nice, smooth line is used to summarize this distribution. Below is an example of a density plot of the Minimum Daily Temperatures dataset.

```
# create a density plot
from pandas import Series
from matplotlib import pyplot
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
series.plot(kind='kde')
pyplot.show()
```

Listing 6.5: Example a Density Plot on the Minimum Daily Temperatures dataset.

Running the example creates a plot that provides a clearer summary of the distribution of observations. We can see that perhaps the distribution is a little asymmetrical and perhaps a

little pointy to be Gaussian. Seeing a distribution like this may suggest later exploring statistical hypothesis tests to formally check if the distribution is Gaussian and perhaps data preparation techniques to reshape the distribution, like the Box-Cox transform.

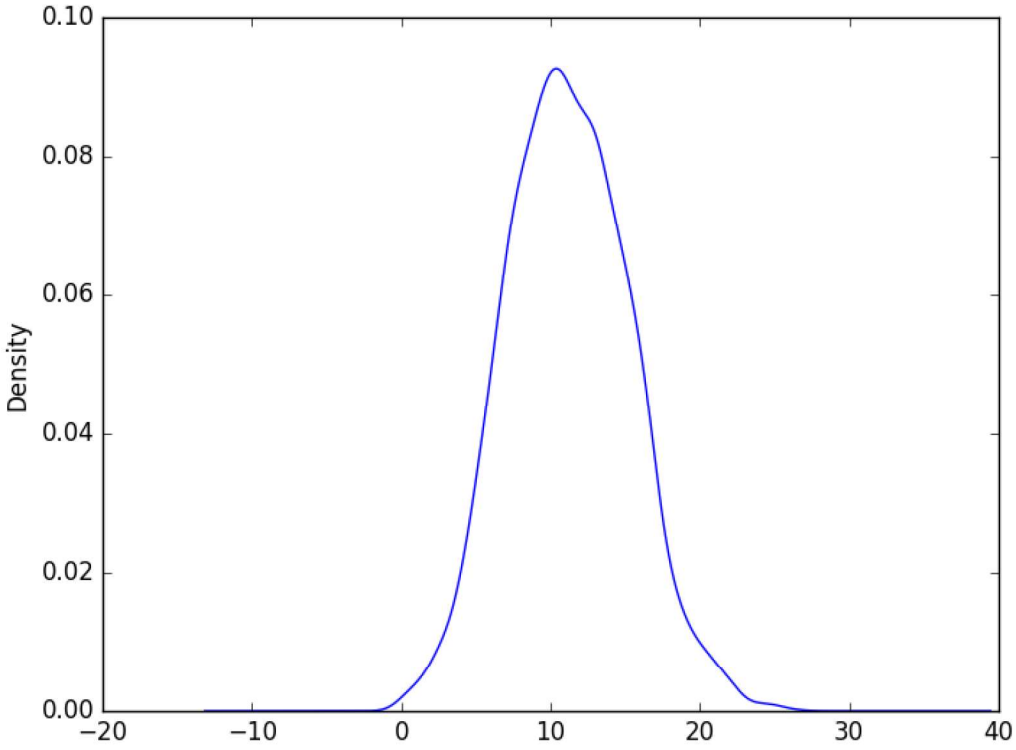

Figure 6.5: Density Plot of the Minimum Daily Temperatures dataset.

6.5 Box and Whisker Plots by Interval

Histograms and density plots provide insight into the distribution of all observations, but we may be interested in the distribution of values by time interval. Another type of plot that is useful to summarize the distribution of observations is the box and whisker plot. This plot draws a box around the 25th and 75th percentiles of the data that captures the middle 50% of observations. A line is drawn at the 50th percentile (the median) and whiskers are drawn above and below the box to summarize the general extents of the observations. Dots are drawn for outliers outside the whiskers or extents of the data.

Box and whisker plots can be created and compared for each interval in a time series, such as years, months, or days. Below is an example of grouping the Minimum Daily Temperatures dataset by years, as was done above in the plot example. A box and whisker plot is then created for each year and lined up side-by-side for direct comparison.

```
# create a boxplot of yearly data
from pandas import Series
```

```
from pandas import DataFrame
from pandas import TimeGrouper
from matplotlib import pyplot
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
groups = series.groupby(TimeGrouper('A'))
years = DataFrame()
for name, group in groups:
  years[name.year] = group.values
years.boxplot()
pyplot.show()
```

Listing 6.6: Example a Yearly Box and Whisker Plots on the Minimum Daily Temperatures dataset.

Comparing box and whisker plots by consistent intervals is a useful tool. Within an interval, it can help to spot outliers (dots above or below the whiskers). Across intervals, in this case years, we can look for multiple year trends, seasonality, and other structural information that could be modeled.

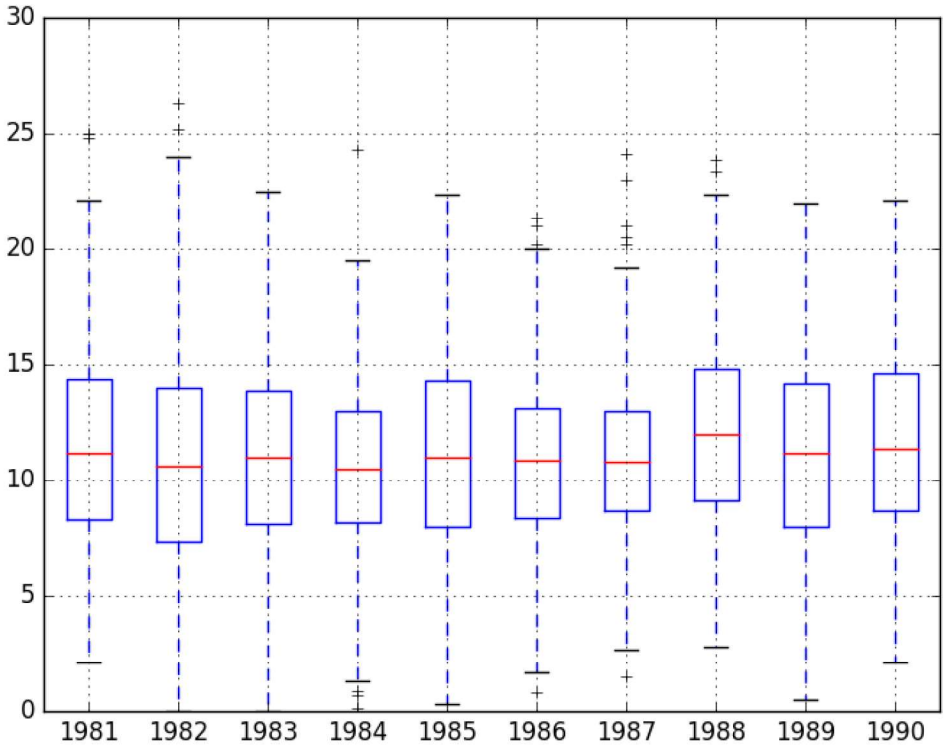

Figure 6.6: Yearly Box and Whisker Plots of the Minimum Daily Temperatures dataset.

We may also be interested in the distribution of values across months within a year. The example below creates 12 box and whisker plots, one for each month of 1990, the last year in the dataset. In the example, first, only observations from 1990 are extracted. Then, the observations

are grouped by month, and each month is added to a new `DataFrame` as a column. Finally, a box and whisker plot is created for each month-column in the newly constructed `DataFrame`.

```
# create a boxplot of monthly data
from pandas import Series
from pandas import DataFrame
from pandas import TimeGrouper
from matplotlib import pyplot
from pandas import concat
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
one_year = series['1990']
groups = one_year.groupby(TimeGrouper('M'))
months = concat([DataFrame(x[1].values) for x in groups], axis=1)
months = DataFrame(months)
months.columns = range(1,13)
months.boxplot()
pyplot.show()
```

Listing 6.7: Example a Monthly Box and Whisker Plots on the Minimum Daily Temperatures dataset.

Running the example creates 12 box and whisker plots, showing the significant change in distribution of minimum temperatures across the months of the year from the Southern Hemisphere summer in January to the Southern Hemisphere winter in the middle of the year, and back to summer again.

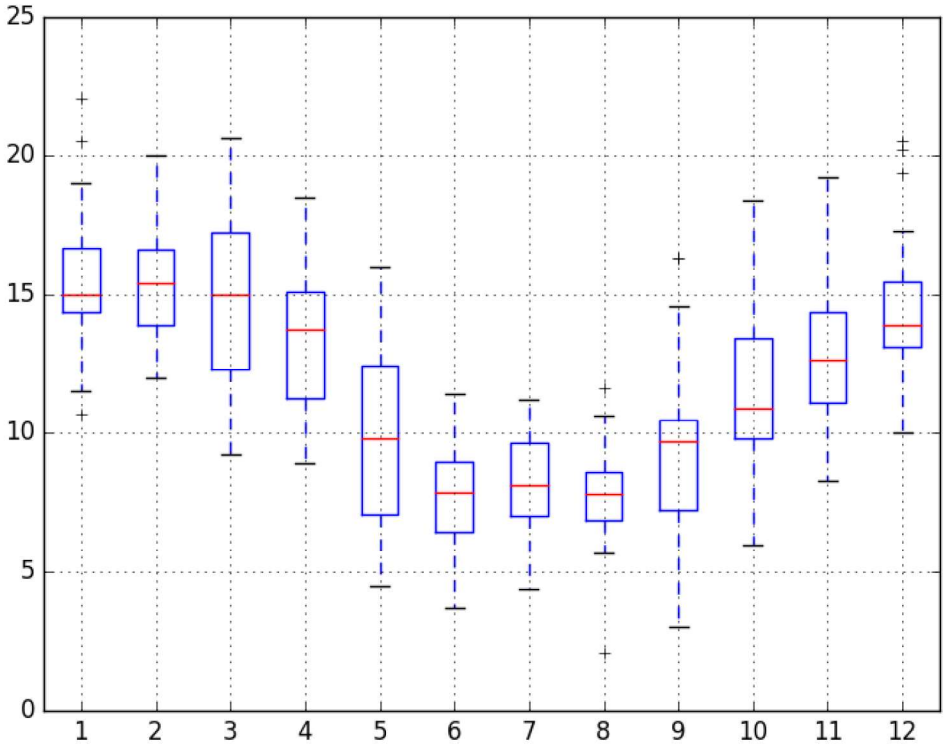

Figure 6.7: Monthly Box and Whisker Plots of the Minimum Daily Temperatures dataset.

6.6 Heat Maps

A matrix of numbers can be plotted as a surface, where the values in each cell of the matrix are assigned a unique color. This is called a heatmap, as larger values can be drawn with warmer colors (yellows and reds) and smaller values can be drawn with cooler colors (blues and greens). Like the box and whisker plots, we can compare observations between intervals using a heat map.

In the case of the Minimum Daily Temperatures, the observations can be arranged into a matrix of year-columns and day-rows, with minimum temperature in the cell for each day. A heat map of this matrix can then be plotted. Below is an example of creating a heatmap of the Minimum Daily Temperatures data. The `matshow()` function from the Matplotlib library is used as no heatmap support is provided directly in Pandas. For convenience, the matrix is rotated (transposed) so that each row represents one year and each column one day. This provides a more intuitive, left-to-right layout of the data.

```
# create a heat map of yearly data
from pandas import Series
from pandas import DataFrame
from pandas import TimeGrouper
from matplotlib import pyplot
```

```
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
groups = series.groupby(TimeGrouper('A'))
years = DataFrame()
for name, group in groups:
  years[name.year] = group.values
years = years.T
pyplot.matshow(years, interpolation=None, aspect='auto')
pyplot.show()
```

Listing 6.8: Example a Yearly Heat Map Plot on the Minimum Daily Temperatures dataset.

The plot shows the cooler minimum temperatures in the middle days of the years and the warmer minimum temperatures in the start and ends of the years, and all the fading and complexity in between.

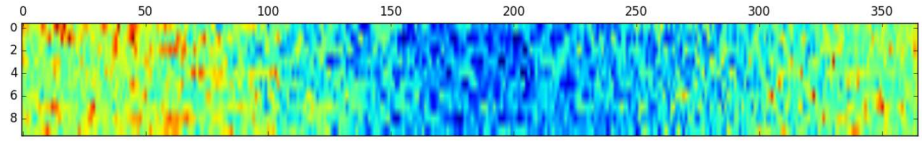

Figure 6.8: Yearly Heat Map Plot of the Minimum Daily Temperatures dataset.

As with the box and whisker plot example above, we can also compare the months within a year. Below is an example of a heat map comparing the months of the year in 1990. Each column represents one month, with rows representing the days of the month from 1 to 31.

```
# create a heat map of monthly data
from pandas import Series
from pandas import DataFrame
from pandas import TimeGrouper
from matplotlib import pyplot
from pandas import concat
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
one_year = series['1990']
groups = one_year.groupby(TimeGrouper('M'))
months = concat([DataFrame(x[1].values) for x in groups], axis=1)
months = DataFrame(months)
months.columns = range(1,13)
pyplot.matshow(months, interpolation=None, aspect='auto')
pyplot.show()
```

Listing 6.9: Example a Monthly Heat Map Plot on the Minimum Daily Temperatures dataset.

Running the example shows the same macro trend seen for each year on the zoomed level of month-to-month. We can also see some white patches at the bottom of the plot. This is missing data for those months that have fewer than 31 days, with February being quite an outlier with 28 days in 1990.

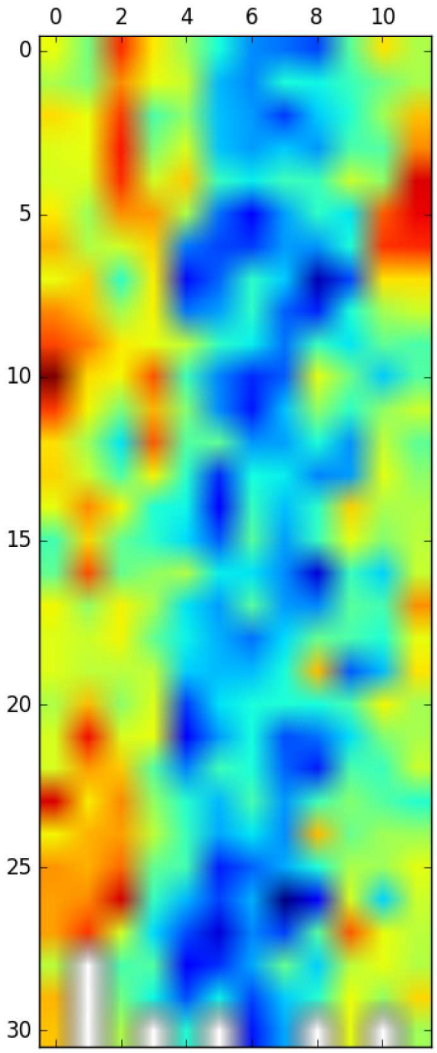

Figure 6.9: Monthly Heat Map Plot of the Minimum Daily Temperatures dataset.

6.7 Lag Scatter Plots

Time series modeling assumes a relationship between an observation and the previous observation. Previous observations in a time series are called lags, with the observation at the previous time step called `lag1`, the observation at two time steps ago `lag=2`, and so on. A useful type of plot to explore the relationship between each observation and a lag of that observation is called the scatter plot. Pandas has a built-in function for exactly this called the lag plot. It plots the

observation at time `t` on the x-axis and the `lag=1` observation (`t-1`) on the y-axis.

- If the points cluster along a diagonal line from the bottom-left to the top-right of the plot, it suggests a positive correlation relationship.
- If the points cluster along a diagonal line from the top-left to the bottom-right, it suggests a negative correlation relationship.
- Either relationship is good as they can be modeled.

More points tighter in to the diagonal line suggests a stronger relationship and more spread from the line suggests a weaker relationship. A ball in the middle or a spread across the plot suggests a weak or no relationship. Below is an example of a lag plot for the Minimum Daily Temperatures dataset.

```
# create a scatter plot
from pandas import Series
from matplotlib import pyplot
from pandas.tools.plotting import lag_plot
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
lag_plot(series)
pyplot.show()
```

Listing 6.10: Example of a Lag scatter plot on the Minimum Daily Temperatures dataset.

The plot created from running the example shows a relatively strong positive correlation between observations and their `lag1` values.

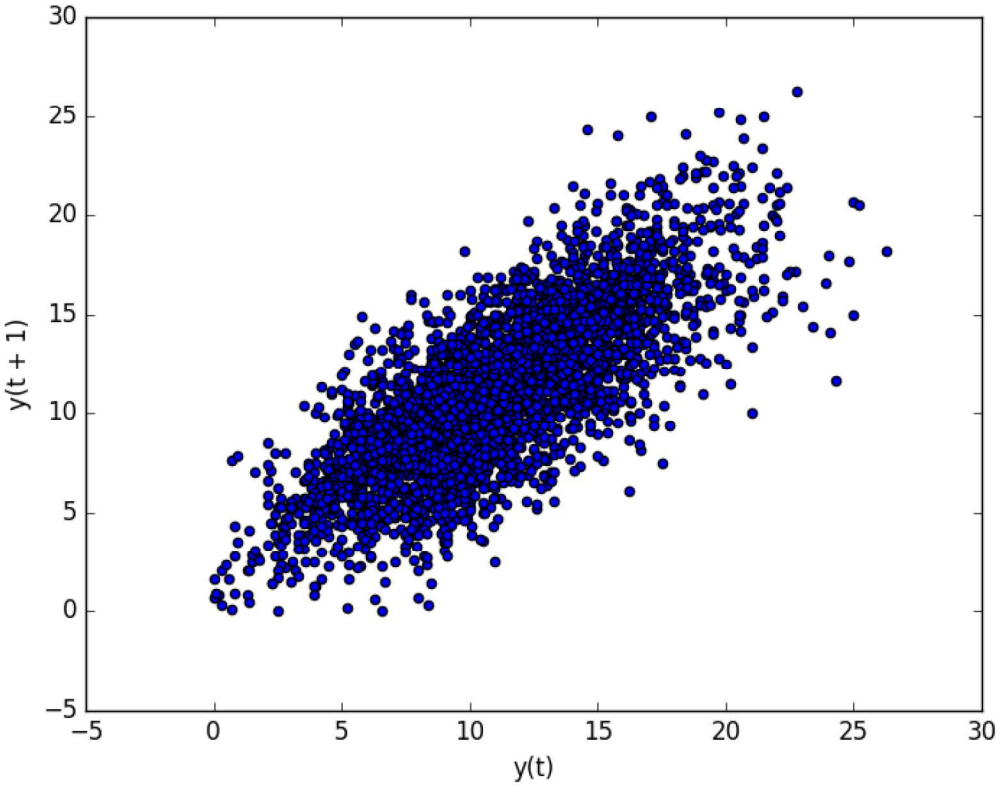

Figure 6.10: Lag scatter plot of the Minimum Daily Temperatures dataset.

We can repeat this process for an observation and any lag values. Perhaps with the observation at the same time last week, last month, or last year, or any other domain-specific knowledge we may wish to explore. For example, we can create a scatter plot for the observation with each value in the previous seven days. Below is an example of this for the Minimum Daily Temperatures dataset. First, a new `DataFrame` is created with the lag values as new columns. The columns are named appropriately. Then a new subplot is created that plots each observation with a different lag value.

```
# create multiple scatter plots
from pandas import Series
from pandas import DataFrame
from pandas import concat
from matplotlib import pyplot
from pandas.tools.plotting import scatter_matrix
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
values = DataFrame(series.values)
lags = 7
columns = [values]
for i in range(1,(lags + 1)):
  columns.append(values.shift(i))
dataframe = concat(columns, axis=1)
columns = ['t+1']
for i in range(1,(lags + 1)):
```

```
  columns.append('t-' + str(i))
dataframe.columns = columns
pyplot.figure(1)
for i in range(1,(lags + 1)):
  ax = pyplot.subplot(240 + i)
  ax.set_title('t+1 vs t-' + str(i))
  pyplot.scatter(x=dataframe['t+1'].values, y=dataframe['t-'+str(i)].values)
pyplot.show()
```

Listing 6.11: Example of Multiple Lag scatter plots on the Minimum Daily Temperatures dataset.

Running the example suggests the strongest relationship between an observation with its `lag=1` value, but generally a good positive correlation with each value in the last week.

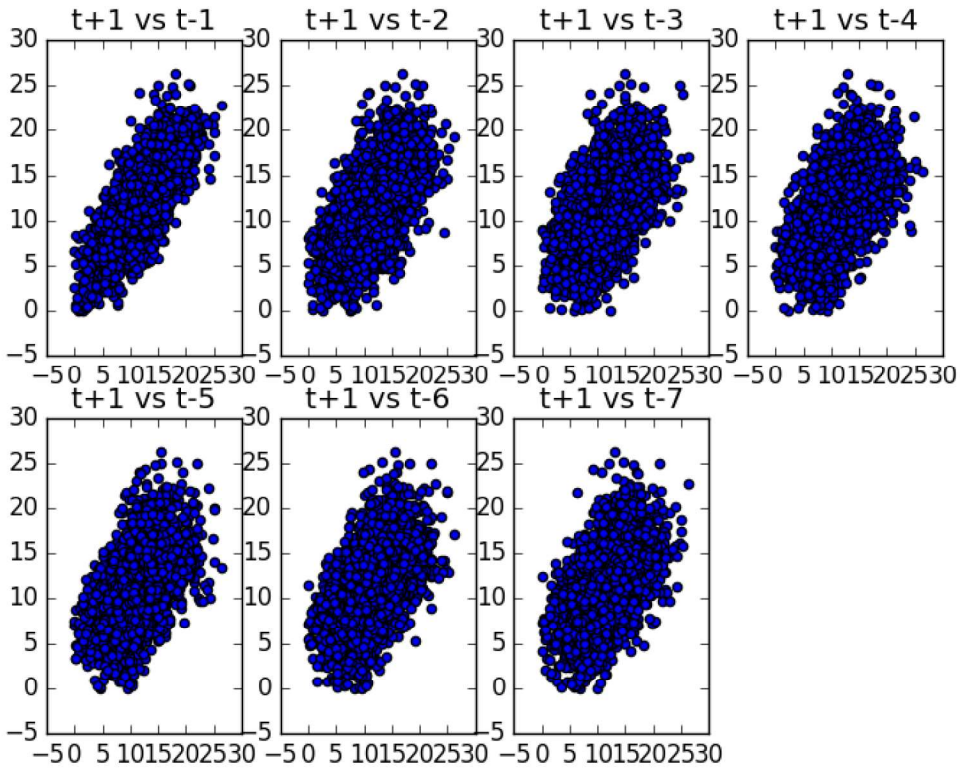

Figure 6.11: Multiple Lag scatter plots of the Minimum Daily Temperatures dataset.

6.8 Autocorrelation Plots

We can quantify the strength and type of relationship between observations and their lags. In statistics, this is called correlation, and when calculated against lag values in time series, it is called autocorrelation (self-correlation). A correlation value calculated between two groups of numbers, such as observations and their `lag=1` values, results in a number between -1 and 1.

The sign of this number indicates a negative or positive correlation respectively. A value close to zero suggests a weak correlation, whereas a value closer to -1 or 1 indicates a strong correlation.

Correlation values, called correlation coefficients, can be calculated for each observation and different lag values. Once calculated, a plot can be created to help better understand how this relationship changes over the lag. This type of plot is called an autocorrelation plot and Pandas provides this capability built in, called the `autocorrelation_plot()` function. The example below creates an autocorrelation plot for the Minimum Daily Temperatures dataset:

```
# create an autocorrelation plot
from pandas import Series
from matplotlib import pyplot
from pandas.tools.plotting import autocorrelation_plot
series = Series.from_csv('daily-minimum-temperatures.csv', header=0)
autocorrelation_plot(series)
pyplot.show()
```

Listing 6.12: Example Autocorrelation Plot on the Minimum Daily Temperatures dataset.

The resulting plot shows lag along the x-axis and the correlation on the y-axis. Dotted lines are provided that indicate any correlation values above those lines are statistically significant (meaningful). We can see that for the Minimum Daily Temperatures dataset we see cycles of strong negative and positive correlation. This captures the relationship of an observation with past observations in the same and opposite seasons or times of year. Sine waves like those seen in this example are a strong sign of seasonality in the dataset.

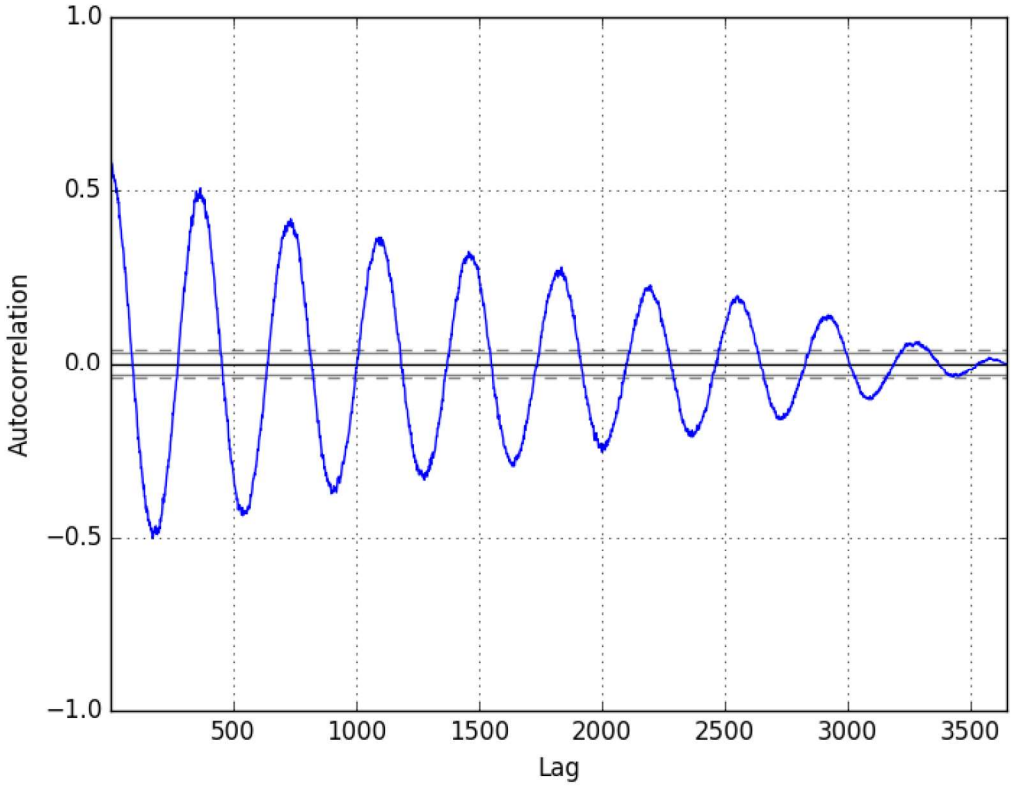

Figure 6.12: Autocorrelation Plot of the Minimum Daily Temperatures dataset.

6.9 Summary

In this tutorial, you discovered how to explore and better understand your time series dataset in Python. Specifically, you learned:

- How to explore the temporal relationships with line, scatter, and autocorrelation plots.
- How to explore the distribution of observations with histograms and density plots.
- How to explore the change in distribution of observations with box and whisker and heat map plots.

6.9.1 Next

In the next lesson you will discover how to change the frequency of 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.

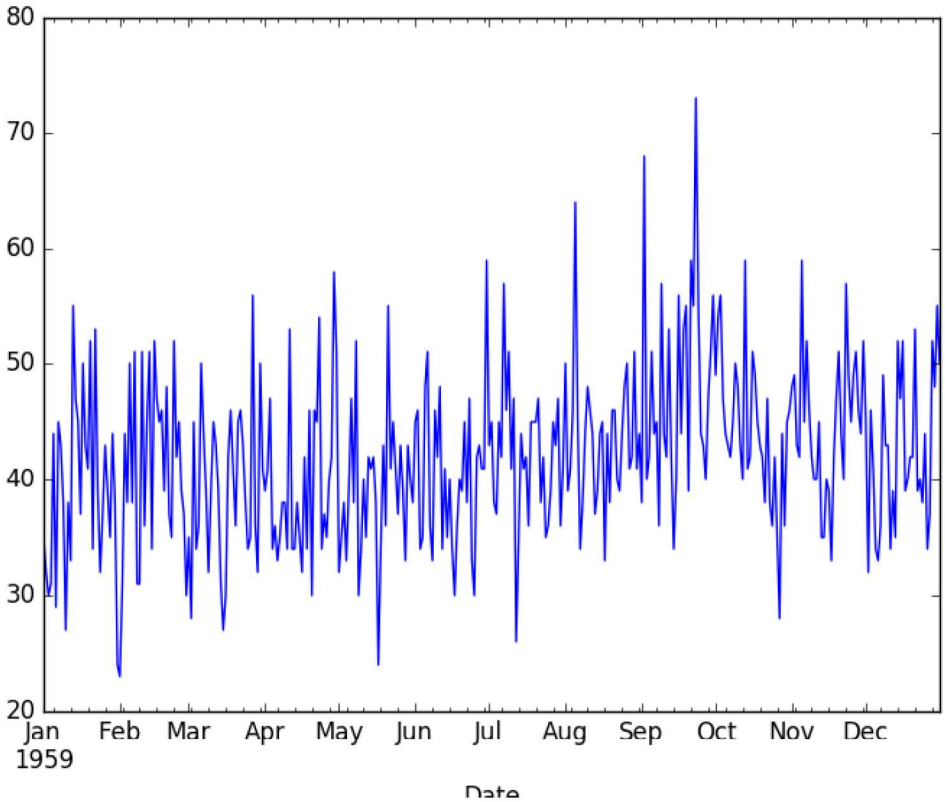

Figure 15.1: Line plot of the stationary Daily Female Births time series dataset.

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.

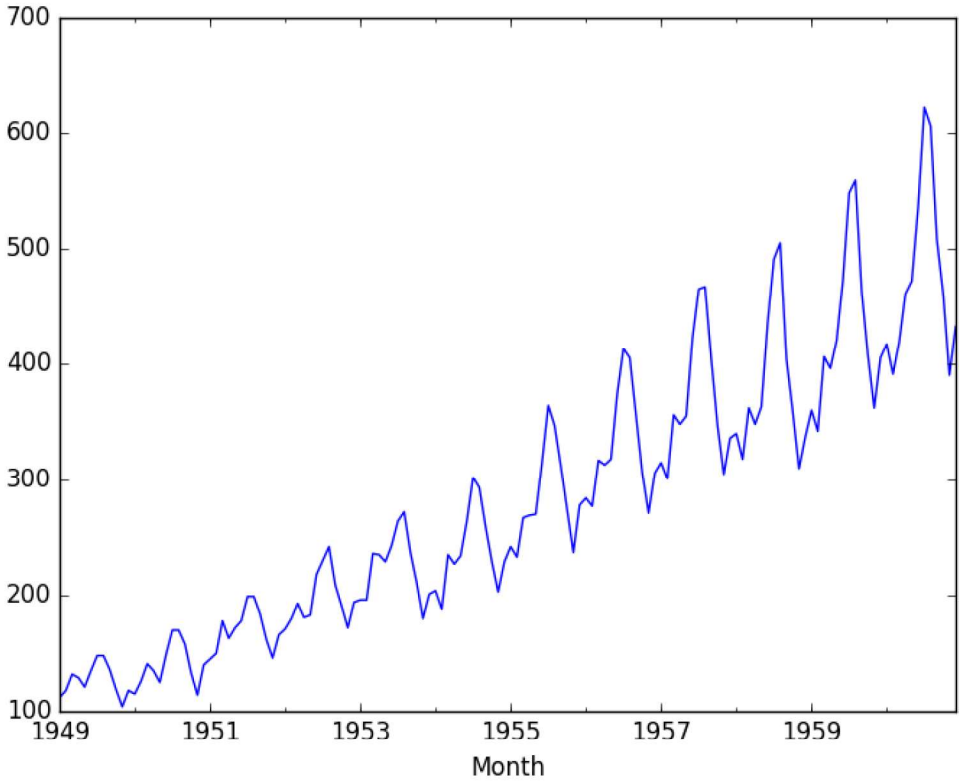

Figure 15.2: Line plot of the non-stationary Airline Passengers time series dataset.

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.

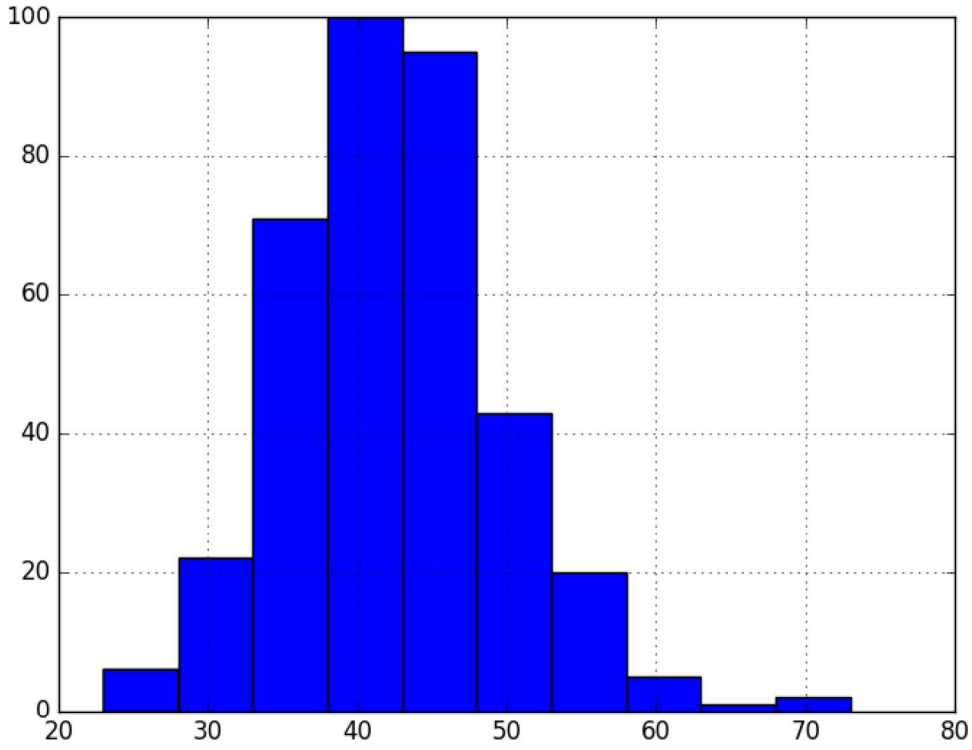

Figure 15.3: Histogram plot of the Daily Female Births dataset.

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.

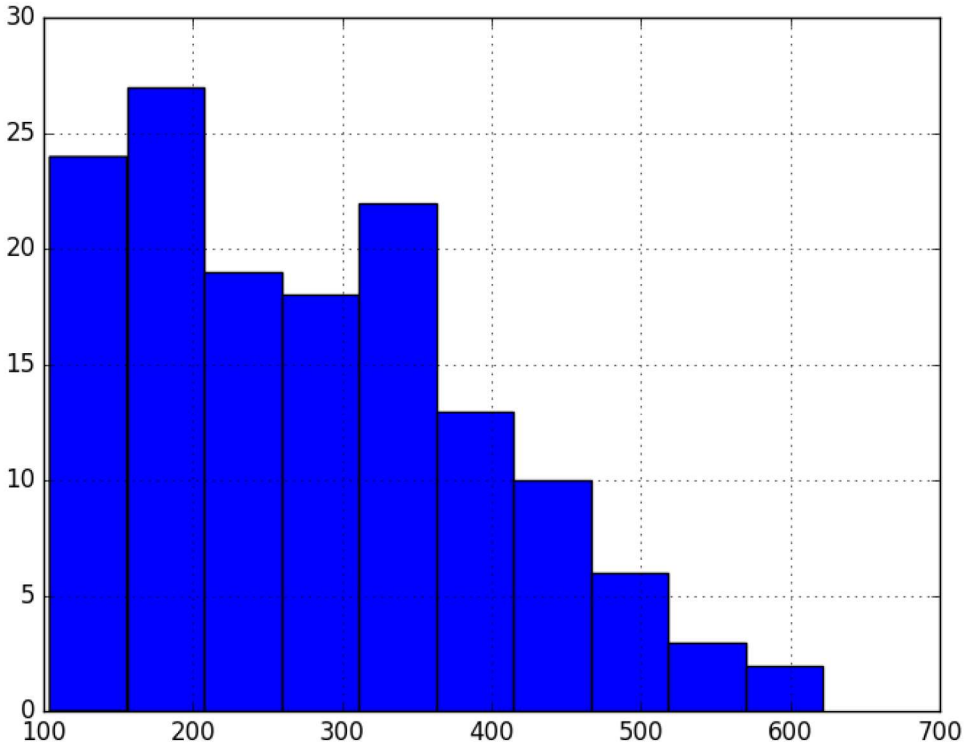

Figure 15.4: Histogram plot of the Airline Passengers dataset.

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.

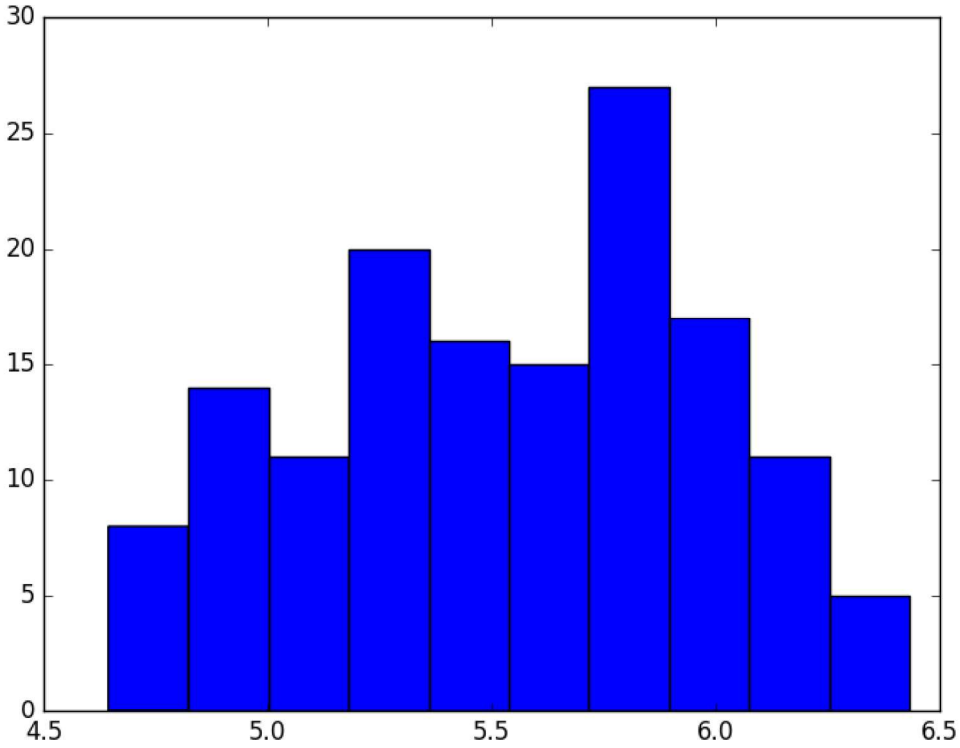

Figure 15.5: Histogram plot of the log-transformed Airline Passengers dataset.

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.

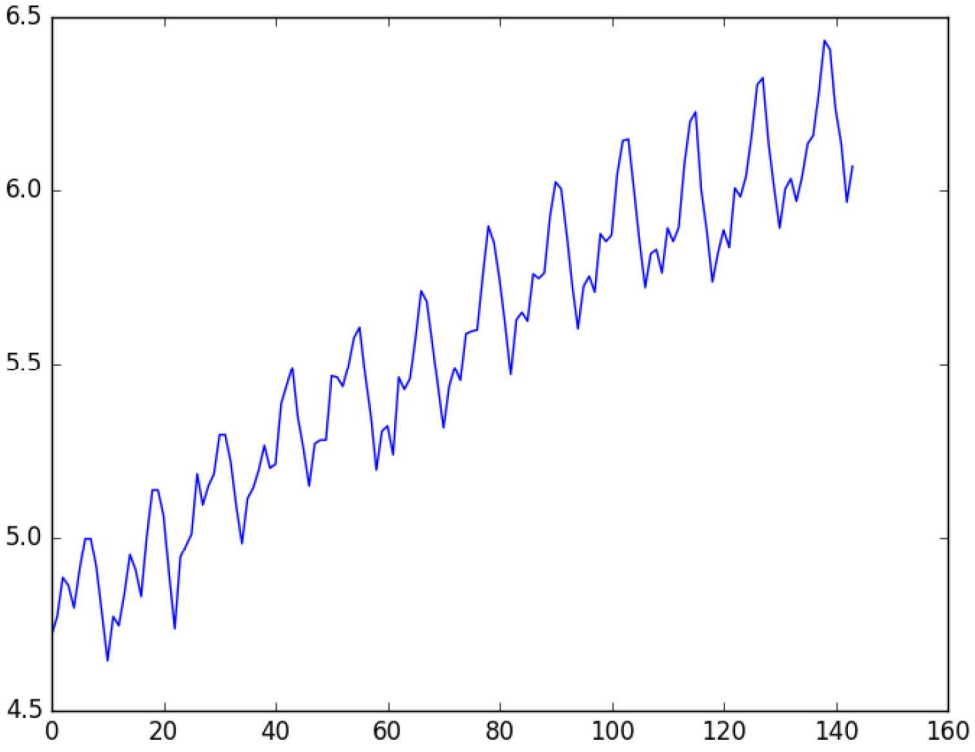

Figure 15.6: Line plot of the log-transformed Airline Passengers dataset.

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.

15.8.1 Next

This concludes Part III. Next in Part IV you will discover how to evaluate models for time series forecasting starting with how to back test models.